APPLE ///

RECORD PROCESSING SERVICES

PROGRAMMING

MANUAL

PRELIMINARY RELEASE
NOVEMBER, 1982

FOR INFORMATION ABOUT RPS, CONTACT
KRIS OLSON
PRODUCT MANAGER
APPLE COMPUTER M/S 22B
5300 STEVENS CREEK BLVD.
SUITE 500
SAN JOSE, CA 95129

(408) 554-5137

@@@@ Preface

This manual describes the Apple III Record Processing Services (RPS), which is a Pascal unit that provides a file-access method for the Apple III.

Before using RPS, you should read the Apple III Owner's Guide. It gives a general introduction to the capabilities and features of the Apple III, and also describes the way the operating system in the Apple III uses files and devices.

If you are not already familiar with the Apple III Pascal system, you should take a look at the manuals that describe the system:

Apple III Pascal: Introduction, Filer, and Editor
Apple III Pascal Program Preparation Tools
Apple III Pascal Programmer's Manual (Volumes 1 and 2)
Apple III Pascal System Technical Reference Manual

These manuals provide all the information necessary to develop and run a Pascal program on the Apple III.

@@@ The Contents of This Manual

Chapter 1 presents an overview of the ideas that form the basis of RPS, including information on file systems in general and RPS in particular.

Chapter 2 contains the details of how to use RPS in a Pascal application program.

Chapter 3 suggests ways to optimize your application to get the most efficient performance possible.

Appendix A is a glossary of all the RPS-related terms used throughout the manual.

Appendix B contains a table of all the possible status returns RPS can issue while an application is executing. The appendix also contains a list of the operating system error codes RPS can return.

Appendix C explains how to repair a damaged RPS file.

Appendix D is a summary of the RPS specifications and restrictions and operating system limitations.

Appendix E contains two sample Pascal applications that use RPS, one short program and one longer, more complex program. Bits and pieces of the complex application are used in Chapter 1 to illustrate some of the concepts of RPS and in Chapter 2 to illustrate how to use the RPS interface.

Appendix F is a listing of the RPS interface.

Contents

Preface i

1 Introduction to RPS 4

4 What Is RPS?
4 Basic RPS Terminology
6 Keyed Access
7 How RPS Works
7 Organizing a File
7 Record Layouts
8 Key Values and Key Definitions
11 More Details on Key Definitions
15 Page Size
15 Summary of "Key" Terms
16 Scanning a File
19 Record Access
21 Record Processing
21 Record Locking
23 Scan Examples
28 Communicating With RPS
28 Access Structures
29 Program Calls
29 The RPS Disk
30 Bibliography

2 Pascal Interface to RPS 31

31 Special Programming Considerations
31 RPS_PTR_LOAD
32 Record and Key Areas
32 File Level Processing
33 Building the Access Structures
35 File Access Structure
38 Key Descriptors
41 Segment Descriptors
48 File Information Structure
50 Key Information Structure
52 Segment Information Structure
54 File Level Calls
54 RPS_CREATE

56 RPS_DESTROY
57 RPS_ERASE
58 RPS_FILEINFO
59 RPS_KEYINFO
60 RPS_SEGINFO
61 Record Level Processing
61 Building the Scan Access Structure
66 Record Level Calls
66 RPS_OPENSCAN
68 RPS_CLOSESCAN
69 RPS_FETCH
71 RPS_POSITION
73 RPS_INSERT
75 RPS_UPDATE
77 RPS_DELETE

3 Guidelines for Memory Initialization and Optimization 79

80 How This Chapter Is Organized
80 Introducing the IS
80 What Does Initialization Mean?
81 Some Key Points About the IS
82 The Layout of the IS
84 Building the IS
84 Space Allocation Components
85 Background Information on Space Allocation
87 key_size
88 record_size
88 num_files
89 num_scans
91 num_indexes
92 num_segs
94 Summary of Space Allocation
95 Buffer Management Components
95 Number of Buffers
96 rel_priority
104 write_threshold
104 extend_size
105 Summary of Buffer Management
106 RPS_INITIALIZE Call
108 Summary Information
108 What You Need to Know About Your Application Before You Specify the IS
110 Summary of Default IS Values
110 Summary of Equations for Space Allocation

Appendexes

A Glossary 113

B Error Messages 118

118 RPS Status Returns
126 SOS Error Messages

C Repairing a Damaged RPS File 129

129 General Information
129 What Is a Damaged File?
130 How to Repair a Damaged File
132 Repairing the File
132 Building the File Repair Access Structures
133 Damage Report Structure
135 Rebuild Structure
137 File Repair Calls
137 RPS_DAMAGE_REPORT
138 RPS_DMG_FILEINFO
139 RPS_DMG_KEYINFO
140 RPS_DMG_SEGINFO
141 RPS_REBUILD

D Summary of Specifications and Restrictions 142

142 RPS Restrictions
143 SOS Restrictions

E Sample Pascal Applications 145

145 Simple Program
148 Complex Program

F RPS Interface 174

RPS Reference Card

@@@@ Chapter 1

@@@ Introduction to RPS

This chapter presents an overview of the ideas that form the basis for the Apple III Record Processing Services (RPS). This chapter

* Defines basic RPS terms.
* Explains generally how RPS works.
* Describes the ways to communicate with RPS and the actions RPS performs.
* Describes what's on the RPS disk and how to add RPS to your system.

@@@ What Is RPS?

RPS is a file management system that operates on the Apple III. In general, the purpose of the RPS file system is to provide a simple way to store information in a file, and subsequently to access and update that information.

@@ Basic RPS Terminology

The basic terminology used with RPS is similar to that used with other file management systems. The following paragraph summarizes this terminology. If you need more information on file management terms, consult the references cited in the bibliography at the end of this chapter.

The smallest unit of information that you manipulate when using RPS is a byte. A specified group of contiguous bytes forms a field, which represents a unit of meaningful information. A group of logically related fields forms a record. Finally, the most inclusive grouping of data is the file itself, which is a collection of related records.

Figure 1-1 shows the compositions of fields, records, and files in general terms and in a specific example.

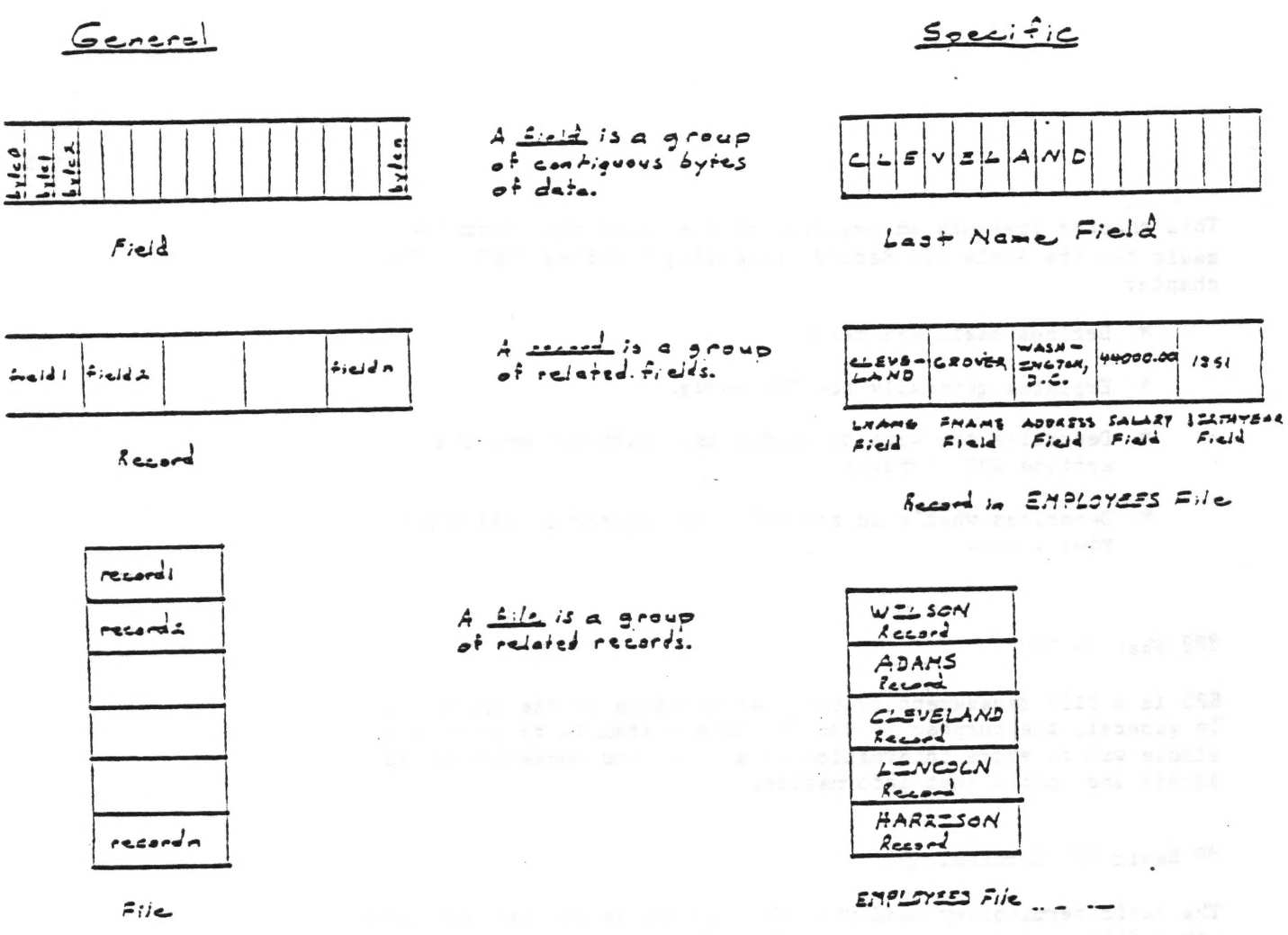


Figure 1-1. Fields, Records, and Files

@@ Keyed Access

This section of the manual uses an example to illustrate the concept of keyed access. If you already know what it means to access a file by key, you can skip this section and proceed to the next section, called How RPS Works, which describes the specific terminology RPS uses for handling keys.

Keyed access is a way of identifying a record, using some of the information contained in that record. The identifying information is called a key. Once you've chosen the key to identify the record, you can get an entire record from the file just by asking for the record with that key.

Example. Suppose you have the following record layout:

last name field	first name field	address field	salary field	birth year field

You might want to use the last name field as a key to find the record of a specific person in the file. You may, in fact, use any one of these fields as keys to locate records within a file.

Suppose you have some records in the file with these names in the last name field.

Plok	...	...	...	...
Vandenburg	...	...	...	...
Jaspers	...	...	...	...
Wetherby	...	...	...	...
Spiegel	...	...	...	...
Gradek	...	...	...	...

To find out Spiegel's salary, all you have to do is ask for the record that has the name Spiegel in the last name field. You'll get the following information:

Spiegel	Tom	Washington, D. C.	35000	1950

@@@ How RPS Works

The operation of RPS involves two basic processes.

* Definition/Creation: the process of defining the way in which records will be organized in an RPS file. This involves establishing a set of rules for RPS on how to handle records. The following section, called Organizing a File, describes this process.

* Manipulation: the process of actually storing, accessing, and updating records in an RPS file. To accomplish this, RPS uses an action called a scan. The section called Scanning a File describes this process.

@@ Organizing a File

The units of information manipulated by the RPS file system are records. Before you create a file, you must first decide on a record layout and then determine in what orders you want to access the records.

@ Record Layouts

The record layout for data records in an RPS file can be either fixed length or variable length. However, records in a single RPS file must be either all fixed length records or all variable length records.

A variable length record has a minimum size and a maximum size. All variable data records must fit within these limits.

Figure 1-2 is a sample record layout, which we will refer to throughout the rest of this manual. It is used in the complex sample application in Appendix E. The record layout represents selected pieces of information about a company's employees.

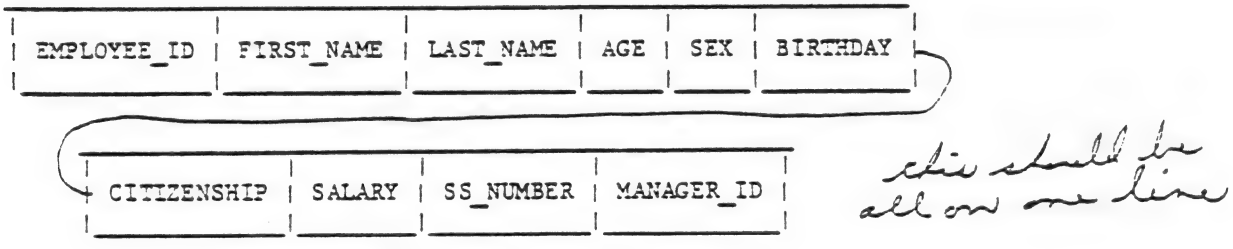


Figure 1-2. Sample Record Layout

@ Key Values and Key Definitions

Introduction. For RPS, the term key has different meanings in different contexts. To keep each meaning as clear as possible, we use a specific word or words in each context. Except for a few rare places, you won't see the word "key" by itself again in this manual.

Key Values. To access a record, RPS must be able to identify the record within the file. RPS does this by designating an area within a record from which it can extract identifying information. The information extracted from each record forms a label that will help to identify the record later. The label is called the key value.

For example, if you have a file of employee records, you might use the portion of the record that contains the employees' ID numbers as the label that identifies each record. The specific number stored in the employee ID area of a record becomes the key value for that record.

Key Definitions. The set of rules that tells RPS how to form a key value is called a key definition. A key definition tells RPS where to look within the record for the information to be used to build the key value. You can have as many as eight key definitions per file.

Indexes. The set of all key values associated with a given key definition for a given file is called the index for that file. RPS sorts the key values in an index according to the order specified in the key definition. The ordering of key values imposes an order on the associated records; this implied ordering of the records is called the sort order of the records for the index. A more detailed discussion of sort order appears later in this chapter.

Thus, in addition to specifying how key values are extracted from records, the key definition also describes how RPS is to order the key values within the index.

Summary. A key definition is the set of rules that tells RPS how to extract information from records to build key values and how to order the key values in the index. Figure 1-3 summarizes the process that occurs when RPS inserts a record into a file.

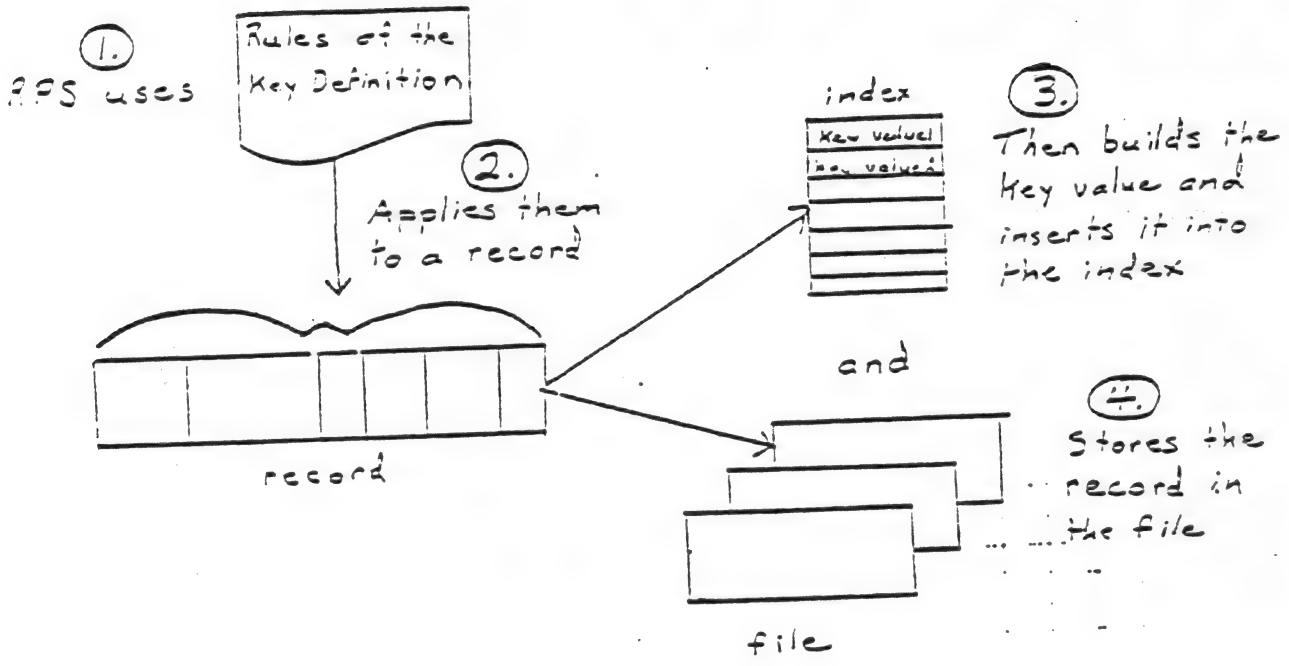


Figure 1-3. Record Insertion Process

In Figure 1-4, you can see a specific example of how key values are extracted from records and placed in an index. This example uses two key definitions for the file of employee records. Each index in the picture contains the ordered set of key values that were extracted from the records according to the appropriate key definition.

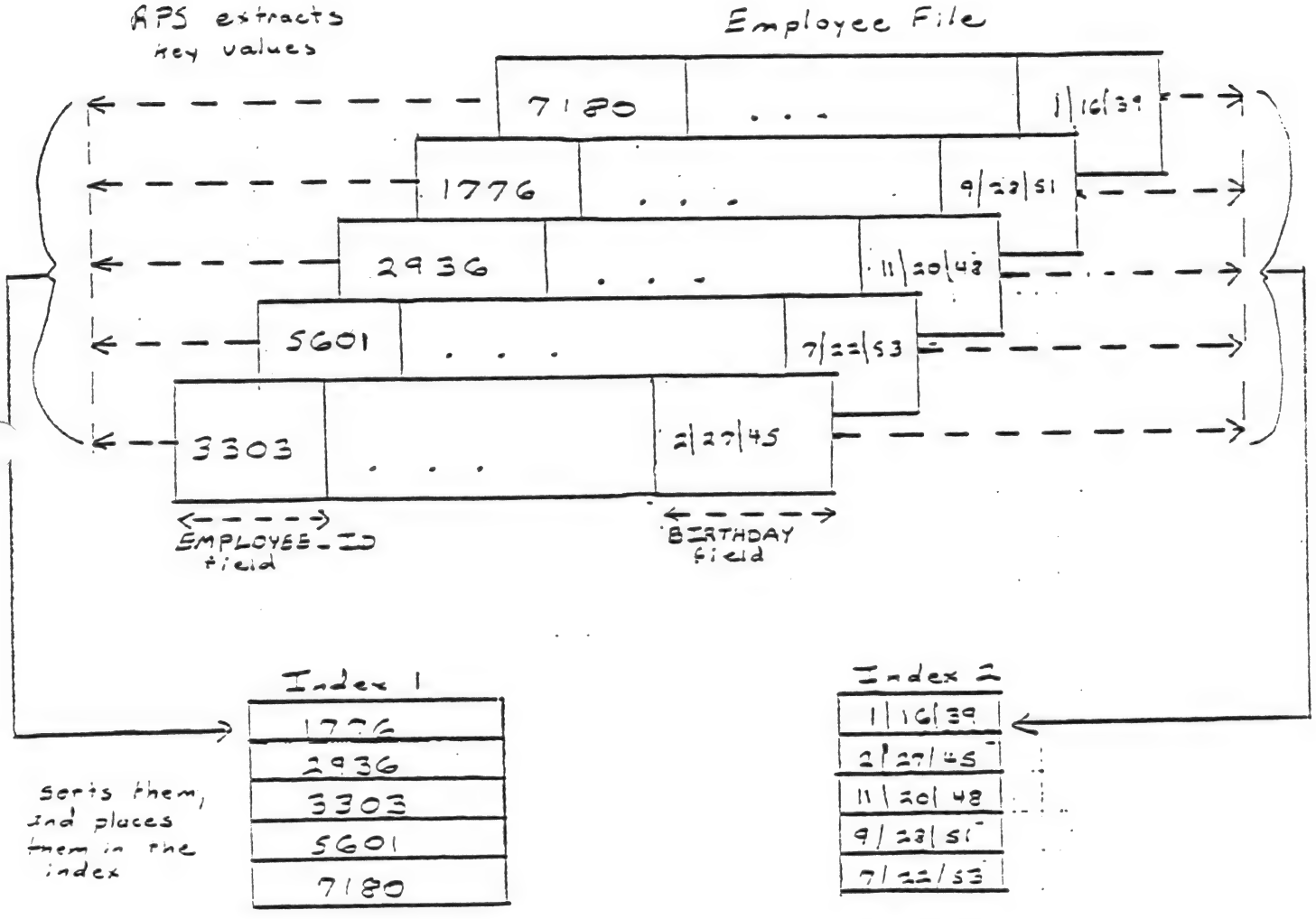


Figure 1-4. Example of How RPS Builds Indexes

Notice that within each index the key values appear in a meaningful order. The employee ID numbers are in ascending numerical order, and the birthdays are in ascending chronological order. The orders for the key values were specified as part of the key definitions. The physical ordering of the records has no effect on the order of the key values in the index.

@ More Details on Key Values and Key Definitions

This section contains more detailed information on the way RPS handles key values and key definitions. Specifically, it describes

Key segments
Null key values
Duplicate key values
Key definition attributes

Key Segments. The portion of a record that you establish as the area from which a key value or part of a key value is extracted is called a key segment. The notion of a key segment is similar to that of a field in that they're both contiguous areas of a record. The difference is in their use. A field holds data, while a key segment holds data to be used in forming a key value. In addition, the boundaries and sizes of key segments do not have to coincide with those of the fields in the record.

When you set up a key definition, you define for RPS the key segments. You may have more than one key segment for a key definition; that is, you may use many areas within the record from which pieces of data for the key value can be extracted.

A single key definition may contain descriptions for as many as eight key segments. However, once RPS forms a key value associated with a key definition, RPS treats the key value as a single entity.

Figure 1-5 shows a record with three key definitions, which illustrate several important characteristics of key segments.

The bytes that compose a single segment have to be contiguous.

The segments that compose a single key definition do not have to be contiguous.

The segments may be located anywhere within the record.

A single key segment may appear in more than one key definition.

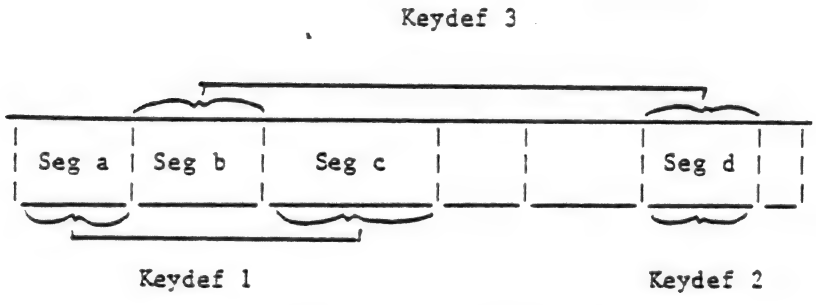


Figure 1-5. Record With Three Key Definitions

Null Key Values. RPS generates a null key value in the following situation: RPS extracts the data from all the key segments of a particular record, as specified in the key definition; collects the values together to form the record's key value; and that value is null. Consult Chapter 2 for a more detailed description of null key values.

When you specify a key definition, you indicate whether or not you want to allow null key values to occur in a record. You need to do this because RPS does not put null key values in an index. If you insert a record that generates a null key value, RPS cannot access the record again using that same index because the index contains no reference to the record.

If you indicate in the key definition that you don't want null key values, RPS will not let you put into the file any record that has a null value for that key value. Nor can you change an existing record so that it contains a null key value.

If you want to be sure that every record in your file is accessible via a certain index, you should specify that null key values are not allowed for that key definition.

Duplicate Key Values. RPS generates a duplicate key value when two records have the same information in every key segment.

When you specify a key definition, you indicate whether or not you want to allow duplicate key values to occur in records. One reason you might want to specify no duplicate values for a particular key definition is to ensure that every key value in the associated index will direct you to one and only one record.

For example, if you're using the employee ID area of an employee record to build key values, it is likely that every employee number will be unique. In this case, you'll want to make sure that every key value is unique, so you should specify that no duplicate values are allowed.

On the other hand, if you're using the birthday area of an employee record to build key values, it is likely that more than one employee will have the same birthday. In this case, you'll probably want to allow duplicate key values.

If you indicate in the key definition that you don't want duplicate key values, RPS will not let you add to the file any record with a key value that duplicates a value already in the index. Nor can you change a record so that it contains a duplicate key value.

If you indicate that duplicate key values are allowed, RPS inserts the duplicate values into the index in the order in which you add them.

Key Definition Attributes. You must specify at least one key definition when you create a file, and every key definition must describe at least one key segment. You build a key definition by specifying the attributes of the key definition as a whole and the attributes of the individual key segments.

The attributes of the key definition are:

* Page size, which is the number of contiguous bytes to be used as a unit for transferring key values between a disk and memory. Page size is described in more detail below.

* A statement of whether or not you want to allow duplicate key values.

* A statement of whether or not you want to allow null key values.

The attributes describing the key segments are

* The starting position of each key segment within the record.

* The length of each key segment.

* The data type of each key segment.

* The order to be used for storing the key values in the index. This order is either ascending or descending. Note that you specify the order for each key segment of a key definition, not for the key definition as a single entity.

Key Value Order Example. Figure 1-6 shows how a change in the order of just one key segment changes the order of the key values in the index.

Portions of Records from a Data File:

EMPLOYEE ID	FIRST NAME	SALARY	SEX
1782	Werner	35,000	m
4529	Maxine	69,000	f
6642	Klaus	63,000	m
0998	Alphonse	14,000	m
1006	Sybil	17,000	f
6666	Gaston	35,000	m
9090	Therese	43,000	f

Key Definitions of Indexes To Be Used:

```
Keydef 1:                        Keydef 2:
     Seg a: field SEX                 Seg a: field SEX
            order ASCENDING                  order ASCENDING
     Seg b: field SALARY              Seg b: field SALARY
            order ASCENDING                  order DESCENDING
     Seg c: field FIRST_NAME          Seg c: field FIRST_NAME
            order DESCENDING                 order DESCENDING
```

Note that the only difference between Keydef 1 and Keydef 2 is the order of the SALARY field.

Key Values Sorted in Index 1:

```
f17000Sybil
f43000Therese
f69000Maxine
m14000Alphonse
m35000Werner
m35000Gaston
m63000Klaus
```

Key Values Sorted in Index 2:

```
f69000Maxine
f43000Therese
f17000Sybil
m63000Klaus
m35000Werner
m35000Gaston
m14000Alphonse
```

Figure 1-6. Key Value Order Example

Explanation:

The ordering of Index 1 is different from that of Index 2 because within each sex group, key values are ordered by salary. For Index 1, the salaries range from lowest to highest; for Index 2, they range from highest to lowest.

The women are first in both indexes because the order is ascending for this segment in both key definitions, and the letter "f" comes before the letter "m" in the alphabet. If the order for this segment were descending, the men would be first.

Note especially that in the case where two men have the same salary, the next segment, FIRST_NAME, determines the order: Werner comes before Gaston in a descending alphabetical ordering.

@ Page Size

A page is a unit of transfer between a disk and memory. With RPS, you can specify how large you want that unit to be for both data records and key values. This allows you to have some control over memory usage and performance.

Note that each index has its own page size; in other words, the index page sizes don't all have to be the same. Also, the index page size doesn't have to be the same as the page size you specify for the data records.

The larger the page size, the more data you have in memory; and the more data you have in the Apple III's memory, the fewer times the Apple III has to retrieve data from the disk. On the other hand, the larger the page size, the more time it takes to transfer each page into memory.

To decide what page size to choose, you must know how records will be accessed. You should experiment with different page sizes to find the ones that provide the most efficient performance for your application. For more information on page size, refer to Chapter 3.

@ Summary of "Key" Terms

Key Definition

A key definition is the set of rules that tells RPS how to extract information from a record to build a key value and how to order the collection of key values in the index.

Key Segment

A key segment is the specific area of a record defined as the area where RPS will find part or all of the key value for the record. Key segments are part of the key definition.

Key Value

A key value is the information RPS extracts from a record to use as an identifying label for that record. RPS uses a key definition to determine where to look for the key value within the record.

Index

An index is the ordered set of all the key values RPS extracts from records in a given file, using a key definition for that file.

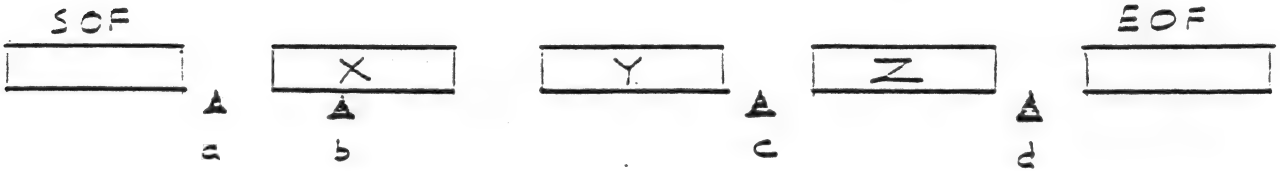
@@ Scanning a File

Accessing information in a file consists of selecting sequences of records. For RPS, this process is called scanning a file. A scan is the action of accessing particular records in some prescribed order.

Think of a scan as a pathway through the records in a file. The path is determined by two factors:

* The index of key values that you use to find the records.
* The way in which you find the records, that is, the sequence of calls and associated access modes.

Current Record of Scan (CROS). A scan has an internal record pointer associated with it. This pointer, called the Current Record of Scan (CROS), can point to four types of locations in a file, as shown in Figure 1-7. With the CROS, RPS maintains a current position within a file that allows you to access and manipulate specific records.



The CROS, represented as ▲, can have any of 4 positions:
a) between start-of-file (SOF) and first record
b) directly on a record
c) between 2 records
d) between last record and end-of-file (EOF)

Figure 1-7. CROS Positions

Initiating a Scan. You must initiate or "open" a scan before RPS can manipulate any records in a file. To initiate a scan, you tell RPS which file to scan and which index of key values to use. The order of the key values in the index establishes the sort order of the records in the file. RPS uses the index to access the records.

A scan always uses one, and only one, associated index. Even if you just want to insert new records, not access existing ones, you must specify an index for the scan. To access records concurrently using different indexes, you must open multiple scans on the same file.

When a scan is opened, RPS sets the CROS at an initial position. The initial position is determined by the access mode at the time the scan is opened. Then, the RPS calls manipulate data, which implicitly moves the CROS around within the file.

Upon opening a scan, RPS assigns an integer to the scan to uniquely identify it. This integer, called the Scan Identifier, is used subsequently to identify the scan.

Scan Example. The following example illustrates how a scan works. Figure 1-8 shows a file with an arbitrary number of records. Scan 1 uses index 1, and begins at record White and ends at record Orange. Successive sequential accesses using index 1 form the path through the file shown by the solid line. Scan 2, which uses index 2, begins at record Tan and ends at record Yellow. This scan forms the path through the file shown by the dashed line.

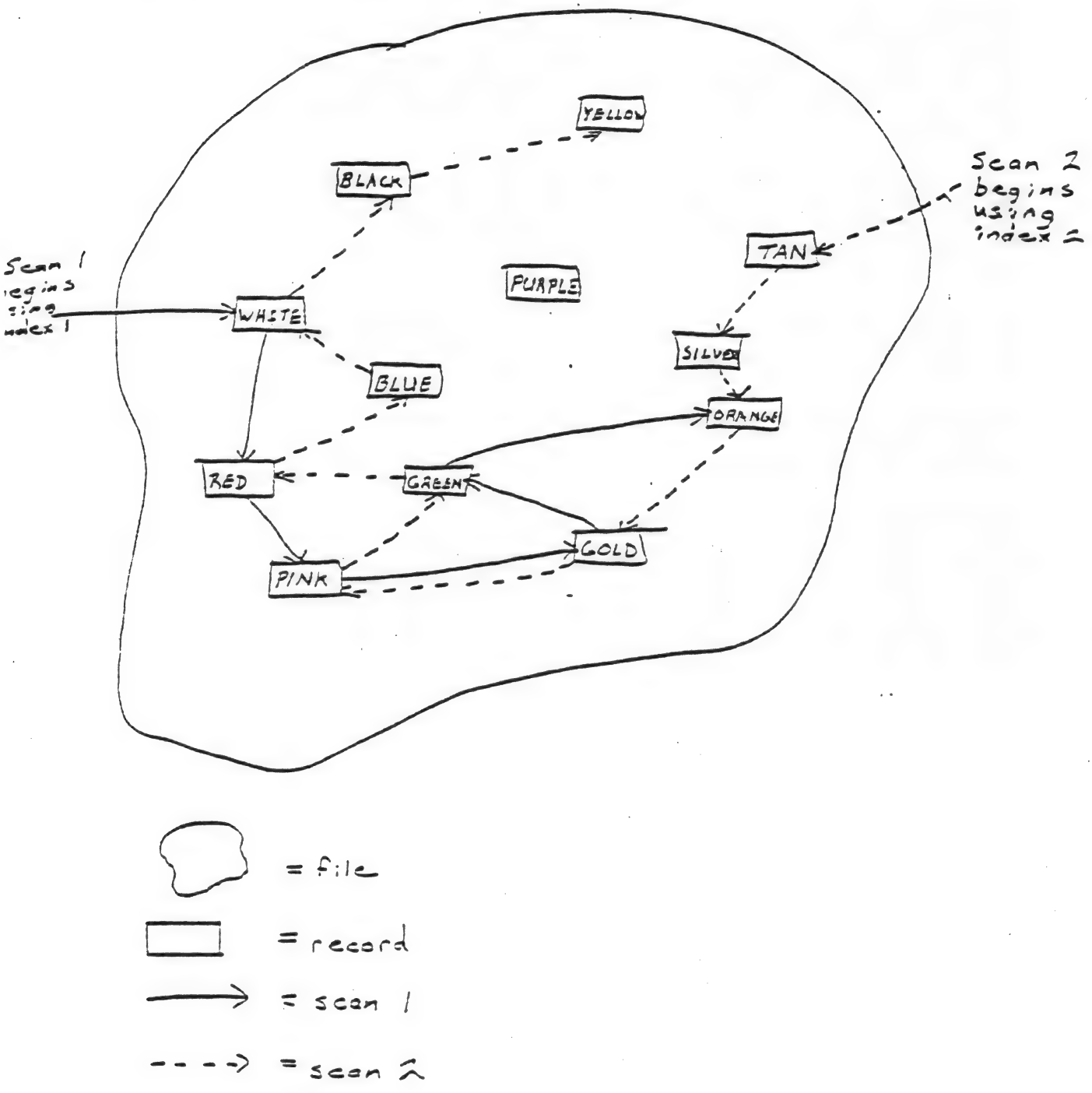


Figure 1-8. Scan Example

Note that record Purple is not touched by either scan 1 or scan 2. This implies one of two things: either record Purple has null values for index 1 and index 2, or the sequence of calls for scan 1 and scan 2 didn't happen to access that record. Other records are touched by one scan and not the other for the same reasons.

Multiple Scans. You can have more than one scan operating on a file at one time. When you use multiple scans for a single file, you can use the same index or different indexes for the various scans, or a combination of both. The scan example above demonstrates multiple scans using different indexes.

@ Record Access

You access records during a scan by issuing RPS calls. To access a record, the CROS must first be moved to the desired record. The positioning of the CROS is determined by the access mode associated with the RPS call. Successive calls can use the same or different access modes.

The following paragraphs describe the six RPS access modes.

Next Mode. When you access a file using Next mode, the record you select is the one following the current record in the sort order, as determined by the position of the CROS. (Remember that the CROS is the pointer that marks your current position in the file.) RPS then updates the CROS to point to the newly selected record.

If you try to use Next mode after you've accessed the last record in the file, RPS positions the CROS between the last record and the end-of-file and gives you a status code indicating that you've reached the end-of-file.

Prior Mode. When you access a file using Prior mode, the record you select is the one preceding the current record in the sort order, as determined by the position of the CROS. RPS then updates the CROS to point to the newly selected record.

If you try to use Prior mode and no prior record exists, RPS positions the CROS between the start-of-file and the first record and gives you a status code indicating that you've reached the start-of-file.

When using Next and Prior modes, remember that the index imposes the sort order on the records.

First and Last Modes. When you access a file using First or Last mode, you position the CROS without selecting any data.

When you use First mode, RPS positions the CROS between the start-of-file and the first record in the file. To determine which record is the first record, RPS uses the sort order of the records based on the index associated with the scan. With First mode, RPS doesn't retrieve record data.

When you use Last mode, RPS positions the CROS between the last record in the file and the end-of-file. To determine which record is the last record, RPS uses the sort order of the records based on the index associated with the scan. With Last mode, RPS doesn't retrieve record data.

Current Mode. With Current mode, you select the record to which the CROS is currently pointing. If the CROS is pointing between two records, no data is returned and a status code to this effect is issued.

Direct Mode. When you use Direct mode, RPS selects the record containing the key value you supply. RPS positions the CROS to point to the newly selected record.

If no record exists with the specified key value, RPS returns a status code indicating this and positions the CROS to point between the records whose key values bracket the specified key value.

If more than one record exists with the specified key value, RPS chooses the first record stored in the file with that key value.

Record Access Diagram. Figure 1-9 shows how Next and Prior modes work. An ascending sort order of the records is assumed, based on the index used for the scan.

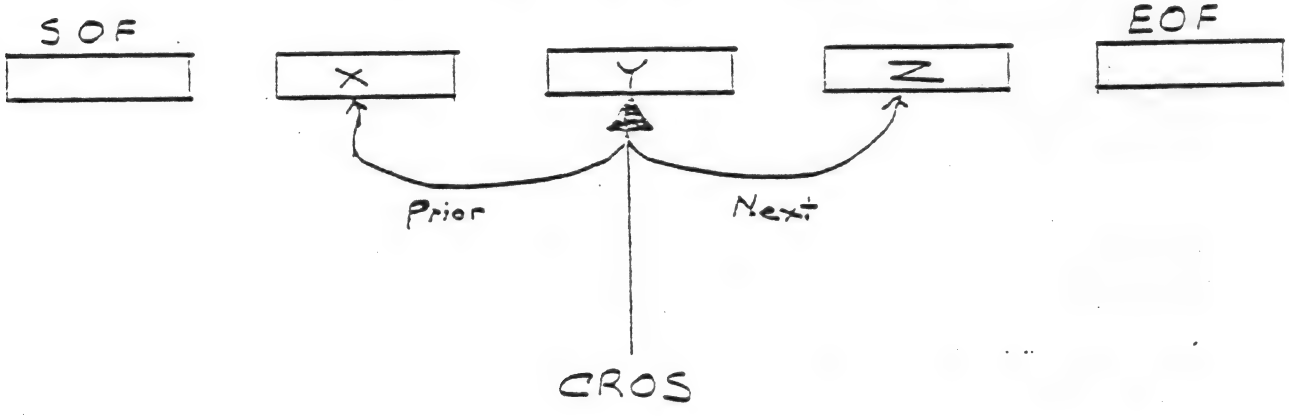


Figure 1-9. Record Access Diagram

@ Record Processing

RPS provides two types of record processing for a scan, read-only processing and modify processing. You specify the type of processing when you initiate the scan, and you cannot change it once the scan is opened.

Read-only processing means that you can get any record in a file during a scan, but you cannot make any changes to the file. Specifically, you cannot change any of the values in a record, you cannot delete any records, and you cannot add any records to the file.

Modify processing means that you can get any record in a file during a scan, and you can make changes to the file. You can add a new record, delete an existing record, and update an existing record.

@ Record Locking

RPS uses a system of record locking when multiple scans are operating. Record locking means that, if a given record is being modified by one scan, it cannot be accessed by any other scan. The record is locked by the scan modifying it.

Record locking is optional. You can turn it on or off when you open a scan and change the setting any time during the scan.

The main intent of the record-locking feature is to provide concurrency control in a multi-user system. In a single-user system, however, this feature is useful in complex applications that have multiple scans open simultaneously for modify processing.

Record locking works as follows: If a scan with record locking turned on touches a record, that record is locked until the scan unlocks the record. The action that occurs when a second scan tries to access the locked record depends on whether the second scan has record locking turned on or off.

> If the second scan has record locking turned on, RPS will not return the record at all. RPS will issue a status return, however, indicating that the record is locked.
>
> If the second scan has record locking turned off, RPS will return the record and you can read its contents. RPS will also issue a status return indicating that the record is in use.

A scan unlocks a locked record in one of two ways.

The scan moves off the locked record to another record.

Record locking for the scan is turned off, and then the scan accesses the record again.

@ Scan Examples

Here are two examples of multiple scans using the sample record presented earlier in this chapter. The first example is of scans using the same index. The second example is of scans using different indexes. The actual programming required to execute these examples appears in the complex sample program in Appendix E.

Example of Multiple Scans Using the Same Index.

Task to Be Accomplished: Print the name and age of every employee and the name and age of every employee's manager.

Relevant Fields from Record:

EMPLOYEE_ID
LAST_NAME
AGE
MANAGER_ID

Index to Be Used: Index of employee ID numbers.

How the Task Is Accomplished:

1. Open two scans, using the index of employee ID numbers for both scans. Use Next mode for Scan 1 and Direct mode for Scan 2.

2. With Scan 1, retrieve the first employee record.

3. Print the employee's name and age.

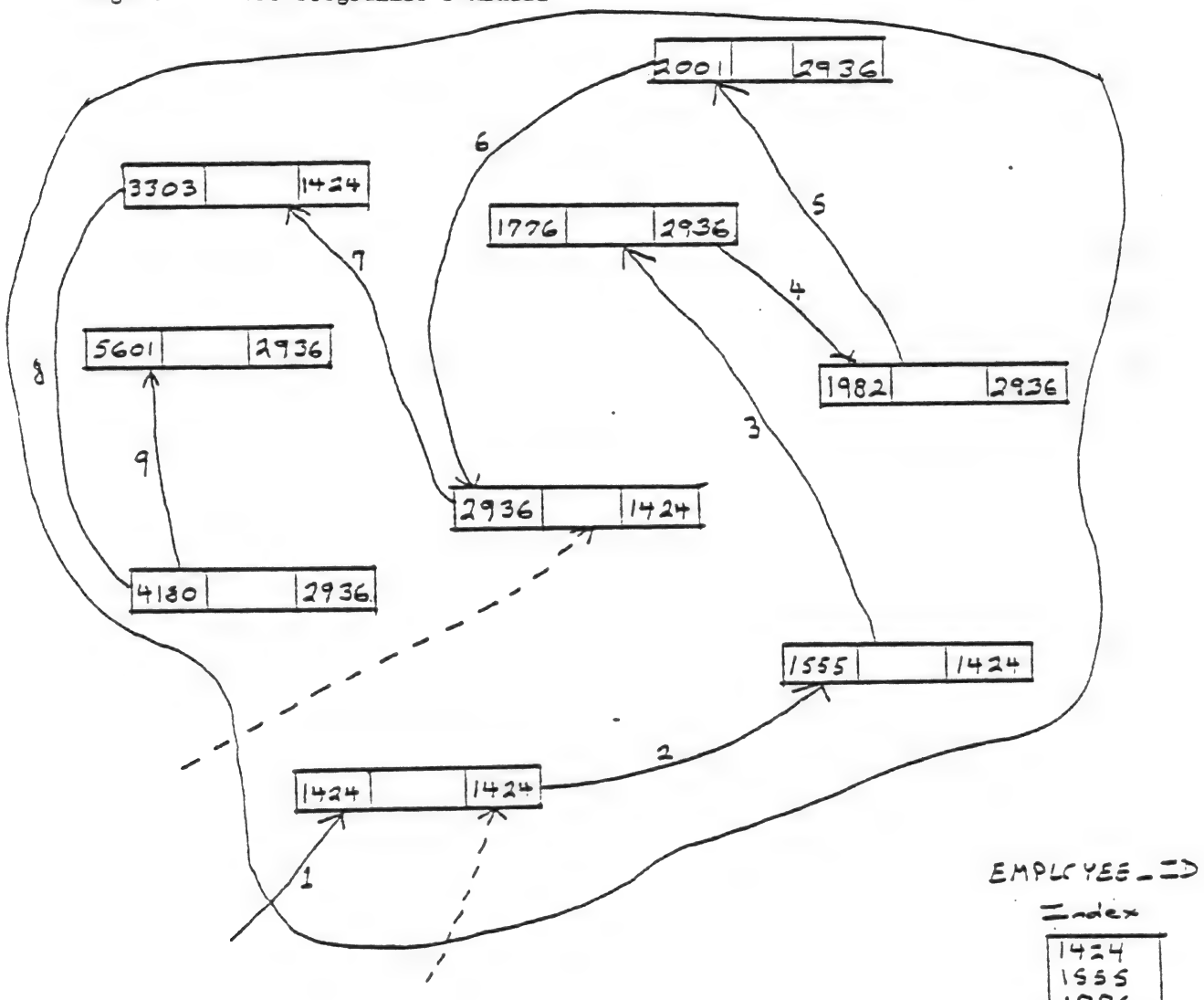
4. With Scan 2, use the manager's ID number from the record just retrieved to find and retrieve that manager's record. (Remember that the manager ID is also an employee ID number.)

5. Print the manager's name and age.

6. With Scan 1 again, retrieve the next employee record and then repeat steps 3, 4, 5, and 6.

   Note that the CROS for Scan 1 will move sequentially through the employee records as you access them, while the CROS for Scan 2 will move directly from one manager record to another.

Graphic Representation: Figure 1-10 shows how the two scans described above operate with real employee ID numbers. Scan 1 touches all records, but Scan 2 touches only the two records of the two managers.



EMPLOYEE_ID Index

1424
1555
1776
1982
2001
2936
3303
4180
5601

Figure 1-10. Example of Multiple Scans Using the Same Index

Example of Multiple Scans Using Two Different Indexes.

Task to Be Accomplished: Print the name and birthday of every employee with a salary equal to or greater than $30,000. For each of these employees, print the names of any other employees who have the same birthday.

Relevant Fields from Record:

LAST_NAME
BIRTHDAY
SALARY

Indexes to Be Used: Index of salary amounts, ordered lowest to highest
Index of birthdays, ordered earliest to latest

How the Task Is Accomplished:

1. Open two scans, using the index of salary amounts for Scan 1 and the index of birthdays for Scan 2.

2. With Scan 1, use Direct mode to find the record of the first employee who earns a salary equal to or greater than $30,000.

3. Change the access mode of Scan 1 to Next.

4. Print the employee's name and birthday.

5. With Scan 2, use Direct mode and the birthday from the record just retrieved to find any record or records with the same birthday.

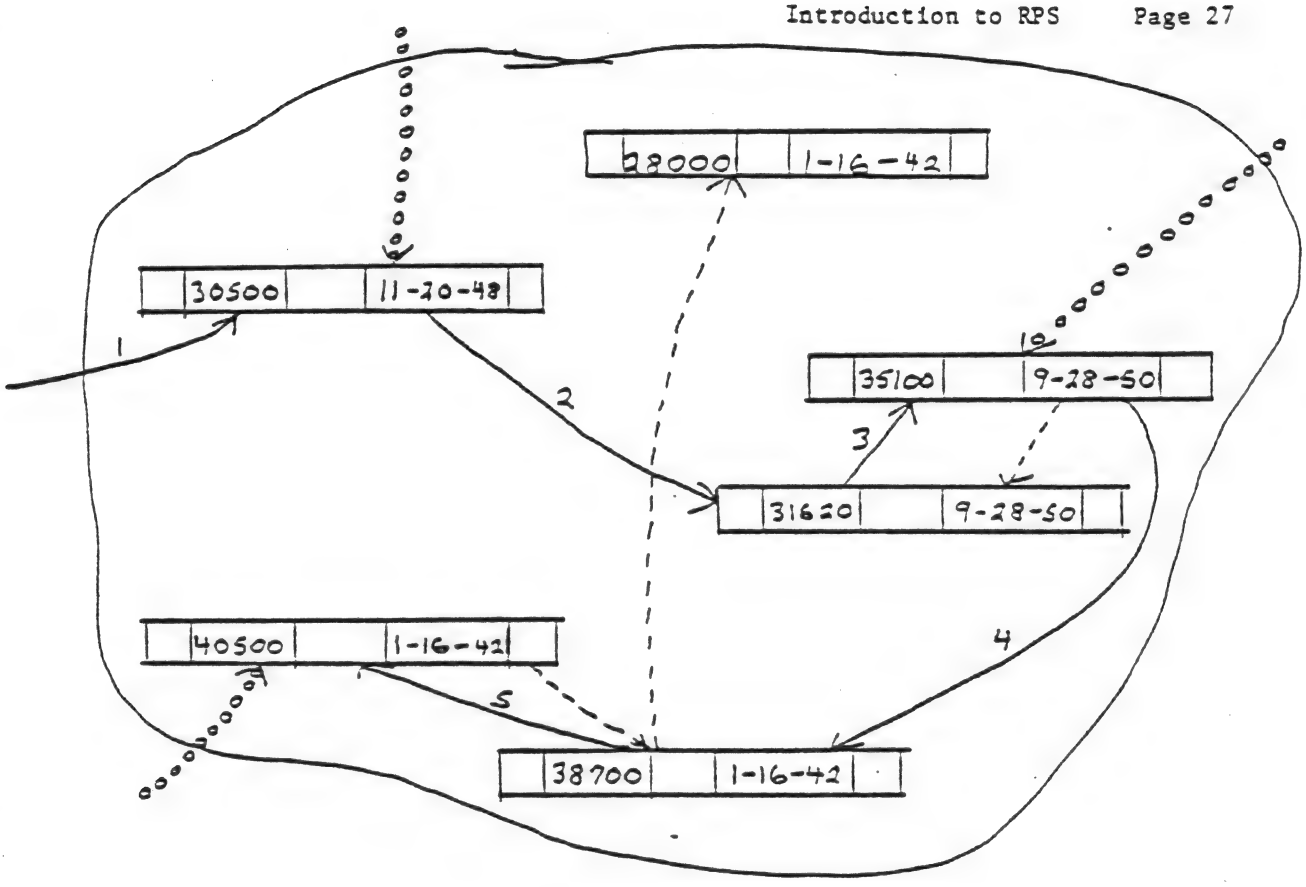
6. Print the name of the employee with that birthday.

7. Still with Scan 2, use Next mode to continue retrieving other records with that birthday, printing names, until you reach a record with a different birthday.

8. With Scan 1 again, retrieve the record of the next employee whose salary is equal to or greater than $30,000, then repeat steps 4, 5, 6, 7, and 8.

   Note that the CROS for Scan 1 will move sequentially through the employee records from the first record with a salary equal to or greater than $30,000 until the end-of-file. The CROS for Scan 2 will move directly to find a specific birthday and then sequentially through all the records with that birthday.

Graphic Representation: Figure 1-11 shows how the two scans described above operate with real salaries and birthdays.



= file

| | X | | Y | |

= record,

where X is SALARY
and Y is BIRTHDAY

SALARY Index

28000
30500
31620
35100
38700
40500

BIRTHDAY Index

1-16-42
1-16-42
1-16-42
11-20-48
9-28-50
9-28-50

——→ = Scan 1

– – – – → = Scan 2 (Next mode)

o o o o o o → = Scan 2 (Direct mode)

Figure 1-11. Example of Multiple Scans Using Two Different Indexes

@@@ Communicating with RPS

The process of communicating with RPS consists of directing RPS to perform a specific action and exchanging information about that action. The means for exchanging information is through data structures called access structures. The means for telling RPS which action to perform is through program calls. The next two sections describe access structures and an overview of program calls. Chapter 2 contains the details of the program calls.

@@ Access Structures

Whenever you request RPS to operate on a file, you must give RPS information about the operation. The way you exchange information with RPS is through access structures.

An access structure is a data structure, which you allocate and maintain in a program, that contains the relevant data for an RPS operation. RPS uses the information you supply in the access structure to execute the operation. RPS also returns information to you via access structures that you provide.

The types of information you exchange with RPS have already been discussed in the previous sections of this chapter. Specifically, when you create a file, you use access structures to communicate

* the structure of the file, including record size and page size for the records
* the key definitions, including the overall attributes and the specific segment attributes

In addition, when you access data, you use an access structure to communicate

* the identifying elements of the scan, including the file to be scanned and the index of key values to be used for the scan
* the kind of record access you want, including the access mode and type of processing allowed for the scan

After RPS executes a command, you can use access structures to obtain information about the action performed and about the structure of the file and key definitions.

RPS uses several different access structures to exchange information for the different file operations. Chapter 2 contains the details of these structures. When you issue a command to RPS, you indicate to RPS the operation you want performed and the appropriate access structure.

@@ Program Calls

You issue commands to RPS in the form of program calls. A call tells RPS which action to perform. As a part of each call, you must specify a reference to the particular access structure that currently contains or will contain the information to be exchanged.

In general terms, the RPS calls perform the following functions:

* Create an RPS file with the organization and attributes that you supply in access structures (RPS_CREATE).
* Delete an existing RPS file (RPS_DESTROY).
* Purge all data from an existing RPS file (RPS_ERASE).
* Retrieve information about the attributes of an RPS file and of a key definition for an RPS file (RPS_FILEINFO, RPS_KEYINFO, and RPS_SEGINFO).
* Begin and end a scan (RPS_OPENSCAN and RPS_CLOSESCAN).
* Retrieve selected records from an RPS file (RPS_FETCH).
* Retrieve selected key values from an RPS file (RPS_POSITION).
* Add records to an RPS file (RPS_INSERT).
* Change selected records in an RPS file (RPS_UPDATE).
* Remove selected records from an RPS file (RPS_DELETE).

@@@ The RPS Disk

The RPS disk contains the following files:

RPS.LIB, with units RPS and RPS_REPAIR	for normal processing
RPS.PAS.CODE RPS.ASM.CODE RPS.UNK.CODE RPS.UNK.TEXT RPS.RPR.CODE	for special processing with the seg_unknown data type

The "special" processing referred to means that you are using the seg_unknown data type and an external assembly language procedure for sorting key values in an index. For an explanation of when and how to use the seg_unknown type, refer to Chapter 2.

Normal Processing. The library file RPS.LIB contains everything you need to use RPS under normal circumstances. It is insufficient only in the special case when you write your own procedure for the seg_unknown data type. Add RPS.LIB to your system as you would any other library file.

Special Processing. When you use the seg_unknown data type and an external procedure for sorting key values, follow these steps to add RPS to your system.

1. Write the specialized assembly language routine to sort the seg_unknown data type. Name the file RPS.UNK.TEXT.

2. Assemble the routine RPS.UNK.TEXT. This will generate the file RPS.UNK.CODE.

3. Link together RPS.PAS.CODE
   RPS.ASM.CODE
   RPS.UNK.CODE

   The result is the RPS unit.

4. Link together RPS.RPR.CODE
   RPS.ASM.CODE
   RPS.UNK.CODE

   The result is the RPS_REPAIR unit.

@@@ Bibliography

The following books contain information on data management systems. The books are listed in order of difficulty from easiest to hardest.

Kroenke, Dave. Database - A Professional Primer. Chicago: Science Research Associates, Inc., 1978.

Martin, James. Principles of Data-Base Management. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1976.

Martin, James. Computer Data-Base Organization, 2nd ed. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1977.

Wiederhold, Gio. Database Design. New York: McGraw-Hill, Inc., 1977.

@@@@ Chapter 2

@@@ Pascal Interface to RPS

This chapter describes the details of RPS use in a Pascal application program. The three main sections in the chapter present the following information:

* Special Pascal programming considerations.
* The program calls and associated access structures needed to process files.
* The program calls and associated access structure needed to process records within a file.

RPS is a Pascal unit. A complete listing of the RPS interface appears in Appendix F.

@@@ Special Programming Considerations

The purpose of this section is to explain the RPS_PTR_LOAD function and to point out certain characteristics of the Pascal language that have special significance in relation to RPS. The topics discussed in this section are specifically related to Pascal and only indirectly related to RPS.

@@ RPS_PTR_LOAD

The RPS_PTR_LOAD function provides a mechanism by which you can pass the location of static variables to RPS. The function returns a pointer to the variable supplied to the function. The variable can be of any type. The call format is

RPS_PTR_LOAD (varname)

The pointer returned is of type ptr_rps_data; the RPS access structures contain some components of this data type. These components are the ones that tell RPS where file names and record and key areas are located in memory. By using the RPS_PTR_LOAD function, you can then load the pointers to the variables in the access structures.

This mechanism allows RPS to avoid prescribing specific types for user records and key values.

HELPFUL HINT: Make sure that the variable to which you are pointing is not a local variable that will be gone by the time the RPS call using it is executed.

@@ Record and Key Areas

Background Information. RPS treats the records and key values in your program as areas of contiguous bytes. The RPS_PTR_LOAD function provides RPS with the starting location of an area. Other parts of the access structures define the sizes of the records and key values.

When RPS acts on any of the record or key areas, RPS looks at exactly the number of bytes you've specified in the access structure for the area, regardless of any Pascal boundaries for data types.

Programming Consideration. Because this is the way RPS handles record and key areas, you must be sure to allocate exactly the same amount of storage space for the record area and key areas as you tell RPS you're going to. If you don't allocate sufficient space, RPS will read from or write into adjacent areas and you will get unpredictable results.

@@@ File Level Processing

RPS provides five program calls for you to use at the file level. The calls are listed below, along with brief descriptions of what they do.

RPS_CREATE	constructs an RPS file without any data
RPS_DESTROY	deletes an RPS file entirely
RPS_ERASE	purges all data from an RPS file, leaving an empty file
RPS_FILEINFO	returns information about a specific file
RPS_KEYINFO	returns information about a specific key definition
RPS_SEGINFO	returns information about a specific key segment

Every call contains one parameter which is a pointer to an appropriate access structure. You allocate and maintain the access structures as part of your program.

@@ Building the Access Structures

RPS uses six access structures at the file level. Each structure contains information relevant to the execution of a file-level operation. The structures are listed below, along with the program calls that use them.

Structure Name	Used In
File Access Structure (FAS)	RPS_CREATE RPS_DESTROY RPS_ERASE
Key Descriptor (KD)	RPS_CREATE
Segment Descriptor (SD)	RPS_CREATE
File Information Structure (FIS)	RPS_FILEINFO
Key Information Structure (KIS)	RPS_KEYINFO
Segment Information Structure (SIS)	RPS_SEGINFO

The FAS, the KD, and the SD are all used to create a file. The FAS points to a list of KDs; each KD in that list points to a list of SDs. Figure 2-1 shows the relationships among these three structures.

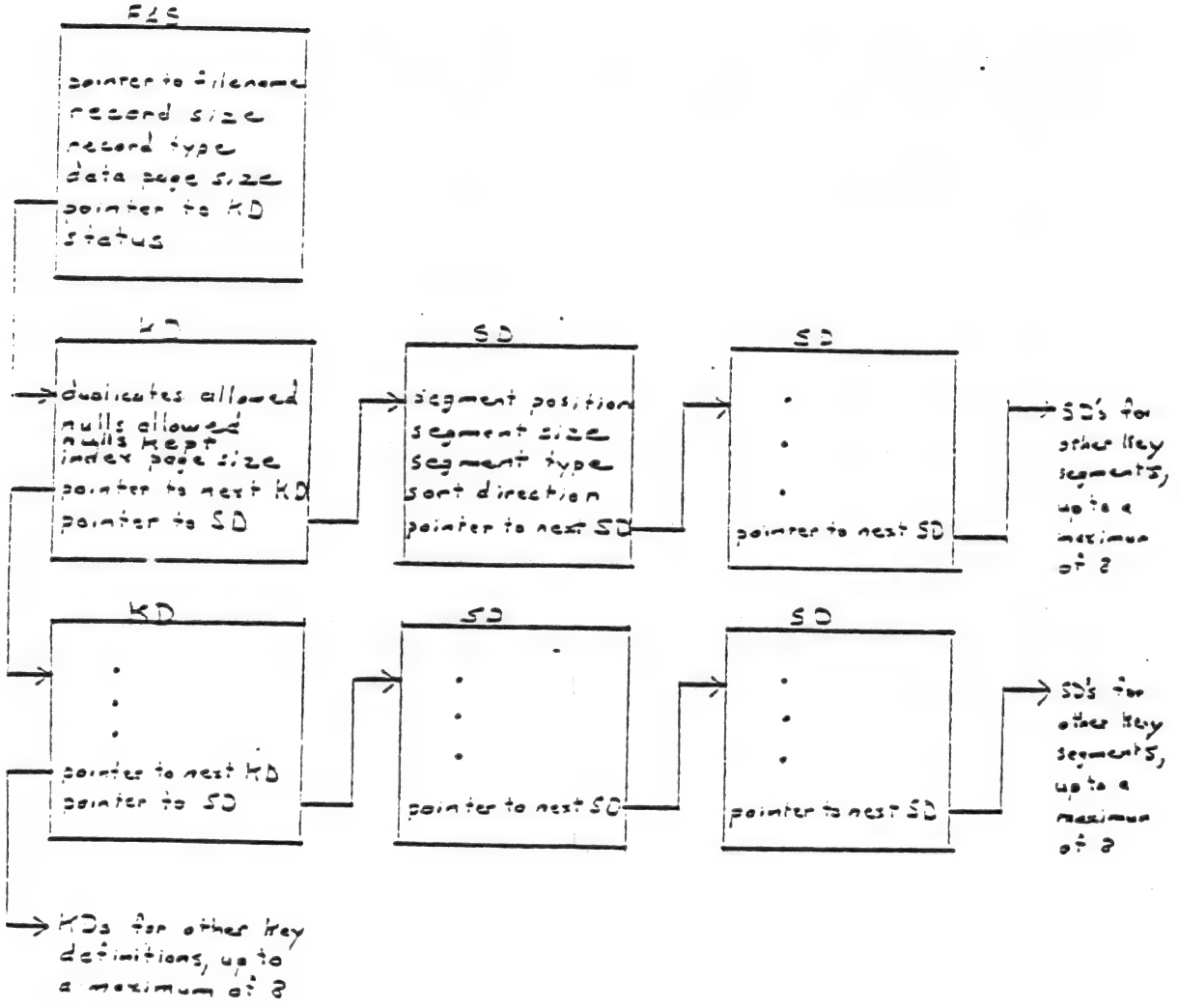


Figure 2-1. Relationships Among The FAS, KD's, and SD's

The following sections contain the details of the access structures RPS uses at the file level.

@ File Access Structure

The File Access Structure (FAS) provides RPS with information about the file you're processing. The FAS is used in the RPS_CREATE, RPS_DESTROY, and RPS_ERASE calls. The layout of the FAS is shown in Table 2-1.

Component	Data Type	Supplied By	Comments
filename_ref	^rps_data	user to RPS	A pointer to a file name area.
record_size	integer	user to RPS	The number of bytes in the record.
record_type	scalar (fixed_length, variable_ length)	user to RPS	The setting that indicates whether the records are fixed or variable length.
data_page_size	integer	user to RPS	The number of contiguous bytes used as a unit for transferring data between memory and a disk.
KD_ref	^KD	user to RPS	A pointer to the first element in a linked list of Key Descriptors for a file.
status	integer	RPS to user	The status of the operation upon completion.

Table 2-1. FAS Layout

Notes on FAS Components: (in bold type)

filename ref

You use the RPS_PTR_LOAD call to fill this component. The file name pointed to can be any valid SOS path name. However the length of the file name of the path name must be two characters less than the maximum allowed by SOS. The maximum file name length allowed by SOS is 15 characters. The maximum path name length is 128 characters.

record size

For fixed length records, record_size is the size of your possible records. The smallest record size is 4 bytes; the largest record size is 2038 bytes. Furthermore, the record size is limited by the data_page_size component; the record size cannot exceed the page size minus 10 bytes.

For variable length records, record_size indicates the maximum record size. RPS will subsequently make sure that no record in the file exceeds that size. The smallest record size is 4 bytes; the largest possible record size is 2035 bytes.

You specify the maximum record size of a variable length record in one of two ways.

You can supply a specific maximum record size. The largest record size is limited by the data_page_size component. To calculate the largest record size, use the following formula. The notation $\lfloor\ \rfloor$ means to take the greatest integer less than or equal to the result of the calculation.

$$\text{max rec size} = \left\lfloor (\text{page size} - 11) - 2 * \frac{(\text{page size} - 11)}{\text{min rec size} + 2} \right\rfloor$$

or

You can specify that the record can be as large as the largest record that can fit on a page. You do this by specifying a record size of 0 . RPS updates the record size to the maximum record size, as computed from the above formula.

Whichever way you use the record_size component for variable length records, RPS will determine the minimum record size. RPS considers the minimum record size to be the length from the beginning of the record to the end of the farthest key segment in the record. All key values are extracted from this area, so the record must be at least this long.

When inserting records into an RPS file, the first two bytes of the record are used to indicate the actual length of the record to RPS.

data page size

RPS sets the data page size to whatever number you specify. The minimum page size is 14 bytes for fixed length records, 17 bytes for variable length records. The maximum page size is 2048 bytes. If you specify a data page size of 0 , RPS uses a page size of 512 .

WARNING: If you set any data page size to be greater than 512 ,you must use the RPS_INITIALIZE call to allocate buffers. For more information, refer to Chapter 3.

Note that the page size for data transfer can be different from the page size for key value transfer.

Number of Records Per Page. To save disk space, RPS packs as many records as possible on each page. When RPS brings a page into memory, it uses buffers whose sizes are multiples of 512 bytes to hold the page.

For fixed length records, the formula for calculating the number of records per page is

$$\text{num recs per page} = \left\lfloor \frac{\text{page size} - 10}{\text{record size}} \right\rfloor$$

For variable length records, the number of records per page depends on the record size. The formula for calculating the space available for records in a page is the same as that for calculating the largest record size:

$$\text{space per page} = \left\lfloor (\text{page size} - 11) - 2 * \frac{(\text{page size} - 11)}{\text{min rec size} + 2} \right\rfloor$$

If every record is the minimum record size, the number of records per page would be

$$\text{num recs if min rec size} = \left\lfloor \frac{\text{page size} - 11}{\text{min rec size} + 2} \right\rfloor$$

FAS Example: (in bold type)

```
new(FAS_ref);
with FAS_ref^ do begin
    filename_ref := RPS_PRT_LOAD(name_of_file);
    record_size := sizeof(data);
    record_type := fixed_length;
    data_page_size := 1024;
    new(KD_ref);
    end;
```

This FAS appears in procedure make_employee_file in the complex sample program in Appendix E.

Key Descriptors

A Key Descriptor, or KD, provides RPS with information about a key definition. Each KD points to a list of Segment Descriptors (SDs), which in turn describe the key segments of the key definition. KDs are used when you're creating an RPS file with the RPS_CREATE call. The layout of a KD is shown in Table 2-2.

Component	Data Type	Supplied By	Comments
dupls_allowed	Boolean	user to RPS	The setting that indicates whether or not duplicate key values are allowed for this key definition.
nulls_allowed	Boolean	user to RPS	The setting that indicates whether or not null key values are allowed for this key definition.
nulls_kept	Boolean	user to RPS	This component is not currently implemented. It should be set to false.
index_page_size	integer	user to RPS	The number of contiguous bytes used as a unit for transferring key values between memory and a disk.
next_KD_ref	^KD	user to RPS	A pointer to the next Key Descriptor in the linked list.
SD_ref	^SD	user to RPS	A pointer to the first element in a linked list of Segment Descriptors for the key definition.

Table 2-2. KD Layout

Notes on KD Components: (in bold type)

dupls_allowed

If you set the dupls_allowed component to true, duplicate key values are allowed in this index. If you set dupls_allowed to false, no duplicate key values are allowed

in the index.

nulls allowed

If you set the nulls_allowed component to true, RPS will let you add records containing null key values; however, such records will not be accessible using this index. If you set nulls_allowed to false, the index will contain a value for every record in the file.

If you set the nulls_allowed component to true, every segment of the key definition must be of the type that can be null. These types are seg_string, seg_packchar, and seg_unknown. A seg_string segment is null if it has length 0 . A seg_packchar segment is null if all bytes in the segment are null characters (ASCII code 0).

A key value is null only if all key segment values are null.

index page size

RPS sets the index page size to whatever number you specify. The minimum page size must be large enough to hold at least four key values plus 30 bytes; the maximum page size is 2048 bytes.

WARNING: If you set any index page size to be greater than 512 , you must use the RPS_INITIALIZE call to allocate buffers. For more information, refer to Chapter 3.

Note that the page size for the transfer of key values in one index can be different from the page size for key values in another index, or for data transfer. Each index has its own page size, and the data has its own page size.

Number of Key Values Per Page. The number of key values per page depends on whether or not duplicate key values are allowed.

If duplicates are allowed, the formula for calculating the number of key values per page is

$$\text{num key values if duplicates} = \left\lfloor \frac{\text{page size} - 14}{\text{key value size} + 6} \right\rfloor$$

If duplicates are not allowed, the formula is

$$\text{num key values if no duplicates} = \left\lfloor \frac{\text{page size} - 14}{\text{key value size} + 4} \right\rfloor$$

next KD ref

You can have a maximum of eight KDs for one file. The last KD in the list must have this component set to NIL.

SD ref

A KD must contain a pointer to a Segment Descriptor.

Additional Miscellaneous Notes on the KD: (in bold type)

Each KD and its associated list of SDs constitute a key definition.

You must have at least one KD for a file when creating the file.

Key Identifier. RPS assigns a unique integer, called a Key Identifier, to each key definition. The Key Identifier has two purposes.

* You use it when you open a scan to identify which index you want to use for the scan.
* You use it when you want to get information about the characteristics of a particular key definition.

Each Key Identifier corresponds to the position of the key definition within the list of KDs. The numbers are assigned sequentially beginning with the number 0 .

KD Example: (in bold type)

```
with current_KD_ref^ do begin
    dupls_allowed := false;
    nulls_allowed := false;
    nulls_kept := false;
    index_page_size := 512;
    new(next_KD_ref);
    new(SD_ref);

    {Segment Descriptors for key definition}

    end;
```

This KD appears in the make_employee_file procedure of the complex sample program in Appendix E.

@ Segment Descriptors

A Segment Descriptor, or SD, gives RPS information about a key segment. Each SD corresponds to one segment of one key definition. SDs are used when you're creating an RPS file with the RPS_CREATE call. You must have at least one SD for each key definition. The composition of an SD is shown in Table 2-3.

Component	Data Type	Supplied By	Comments
pos_in_ record	integer	user to RPS	The byte position within the record where the segment begins.
seg_size	integer	user to RPS	The number of bytes in the segment.
seg_type	scalar (seg_ints16, seg_intu16, seg_ints32, seg_intu32, seg_ints64, seg_intu64, seg_real32, seg_real64, seg_string, seg_packchar, seg_packbyte, seg_unknown)	user to RPS	The data type of the segment.
sort_ direction	scalar (ascending, descending)	user to RPS	The sort direction of the segment.
next_SD_ref	^SD	user to RPS	A pointer to the next Segment Descriptor in the linked list.

Table 2-3. SD Layout

Notes on SD Components: (in bold type)

pos in record

For unpacked records, if you want to use an entire field or the first part of a field as a segment, you can use the Pascal SIZEOF function to determine how much space preceding fields take up. The position of a field in a record is the sum of the preceding field sizes. The first field has position 0 .

If you want to use a portion of a field as a segment, you can use this same technique and then add the additional bytes necessary to specify the proper position.

For segments of type seg_string, the segment position must coincide with the length byte of the string.

To figure the segment positions for packed records, read the section in the Apple III Pascal System Technical Reference Manual that describes the way Pascal allocates memory.

seg size

For all data types except seg_string, seg_packchar, seg_packbyte, and seg_unknown, RPS ignores any value you supply and selects an appropriate size for the data type.

For unpacked records, if you want to use an entire field of the type seg_string, seg_packchar, seg_packbyte, or seg_unknown as a segment, you can use the Pascal SIZEOF function to determine the actual allocated size of the field. If you want to use a portion of this type of field as a segment, you can specify the number of bytes for the segment that you want; however, you should be aware of the way Pascal allocates storage and take care to specify a proper size.

To figure the segment sizes for packed records, read the description in the Apple III Pascal System Technical Reference Manual that describes the way Pascal allocates memory.

seg type

The segment types are

seg_ints16	signed integer 16 bits long
seg_intu16	unsigned integer 16 bits long
seg_ints32	signed integer 32 bits long
seg_intu32	unsigned integer 32 bits long
seg_ints64	signed integer 64 bits long
seg_intu64	unsigned integer 64 bits long
seg_real32	real value 32 bits long
seg_real64	real value 64 bits long
seg_string	string value
seg_packchar	packed array of character
seg_packbyte	packed array of byte
seg_unknown	special type, described below

The segment type seg_unknown is a special type that allows you to supply external assembly procedures to determine the ordering of key values and null key values for this type. For more information, refer to the next section, Additional Miscellaneous Notes on the SD.

next SD ref

You can have a maximum of eight SDs per key definition. The last SD in the list must have this component set to NIL.

Additional Miscellaneous Notes on the SD: (in bold type)

Null Values. The only segment data types that can have null values are

type seg_string, when the length is 0
type seg_packchar, when each character's ASCII value is 0
type seg_unknown, determined by an external procedure.

A key value is null only if all key segment values are null. A null key value means that no access to the record exists for the associated index.

Segment Identifier. RPS assigns a unique integer, called a Segment Identifier, to each key segment defined. The Segment Identifier is the number that you use to identify which segment you want information about when using the RPS_SEGINFO call.

Each Segment Identifier corresponds to the position of the segment definition within the list of SDs. The numbers are assigned sequentially beginning with the number 0 .

Segment Type seg_unknown. The segment type seg_unknown is a special type that lets you define your own specialized sort sequence. To use this type, you must write an external assembly language procedure.

Here's an example of a situation where you might want to use the seg_unknown type.

Suppose you want dates to be stored in the format month/day/year with the following Pascal type:

```
date = packed record
          month : 1..12;
          day : 1..31;
          year : 00..99;
          end;
```

The memory layout is as follows:

The three lists of dates shown below are: the initial dates to be sorted, the dates sorted using type seg_intul6, and the dates sorted chronologically.

Initial Dates	Sorted Using seg_intul6	Sorted Chronologically
5/18/55	7/11/45	7/11/45
7/11/45	8/7/55	3/23/55
3/23/55	5/18/55	5/18/55
12/9/57	3/23/55	8/7/55
1/29/57	12/9/57	1/29/57
8/7/55	1/29/57	12/9/57

There are two ways to have RPS sort the dates chronologically.

* Change the Pascal type definition to

```
date = packed record
          day : 1..31:
          month : 1..12;
          year : 00..99;
          end;
```

* Write an assembly language routine to sort the dates properly.

Although this problem has been solved by reordering the fields in the Pascal record, you might have a sort sequence that is not so easily solved. If this is the case, you may want to use the seg_unknown data type. Consult your Apple dealer for further technical information on the seg_unknown type.

Key Value Comparisons. RPS compares key values one segment at a time. The comparison starts with the segments whose Segment Identifiers are 0 , and continues until a difference in values is found. Once RPS finds an inequality, it uses the sort directions specified in the SDs of the segments to determine which segment comes first.

Consider the example where two key values are equal through the first three segments and unequal at the fourth. The fourth segment of each key value is of type seg_real32 and has a descending sort direction. The key values are

```
Key Value 1, Seg ID 3:        143.78
Key Value 2, Seg ID 3:        2.73E18
```

In this case, Key Value 2 comes before Key Value 1.

Comparisons of the numeric data types occur as you would expect, with the exception listed below for seg_real32 and seg_real64. The comparisons for seg_packbyte, seg_string, and seg_packchar are described below.

seg real32 and seg real64 values

For these types, key values that are Not a Number (NaNs) compare greater than all other key values. For information on NaNs, consult the Apple III Pascal Programmer's Manual (Volume 2).

seg packbyte values

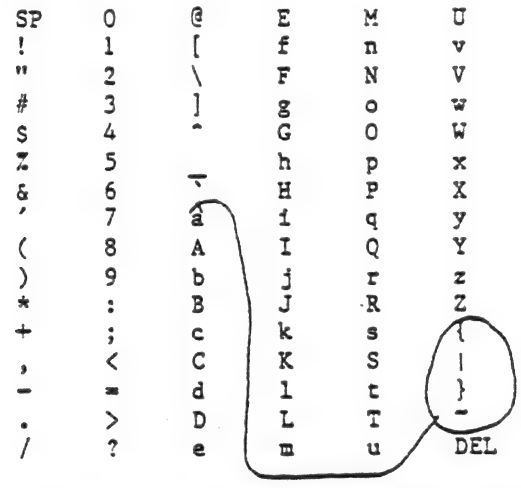
The comparisons of this type are based on the values stored in each byte. For example, here are two values with an ascending sort direction.

	Bytes				Characters
Value 1:	65	66	90	68	ABZD
Value 2:	65	66	97	68	ABaD

Value 1 would come before Value 2. If the bytes were interpreted as ASCII characters, they would have the values shown at the right.

seg_string and seg_packchar values

When comparing segments of these types, RPS considers the order of characters to be the following:



The following example shows how RPS would sort three character sequences:

Unsorted Values		Sorted Values	
Value 1:	ABC	Value 2:	abc
Value 2:	abc	Value 3:	ABc
Value 3:	Abc	Value 1:	ABC

When the values being compared for segments of type seg_string have different lengths, the comparison works like this: RPS first compares the number of characters of the shorter string. If the values are still equal after that comparison, RPS then looks at all the remaining characters in the longer string. If all remaining characters are spaces (ASCII code 32 decimal), RPS considers the two values to be equal. If all remaining characters are not spaces, RPS considers the shorter value to come before the longer value. Here's an example showing how this works for two values with an ascending sort direction.

```
Value 1:      ABC
Value 2:      ABC  D
```

Value 1 would come before Value 2.

For segments of type seg_packchar, all characters of the segment are used.

Example of Key Definition with One SD: (in bold type)

```
with SD_ref^ do begin
    pos_in_record := 0;
    seg_size := sizeof(integer);
    seg_type := seg_ints16;
    sort_direction := ascending;
    next_SD_ref := nil;
    end;
```

Example of Key Definition with Multiple SDs: (in bold type)

```
with SD_ref_^ do begin
    pos_in_record := sizeof(data) -
                     sizeof(integer) -
                     sizeof(number);
    seg_size := 3;
    seg_type := seg_packbyte;
    sort_direction := ascending;
    new(next_SD_ref);
    with next_SD_ref^ do begin
        pos_in_record := sizeof(integer) + 2*sizeof(name);
        seg_size := sizeof(integer);
        seg_type := seg_intu16;
        sort_direction := descending;
        new(next_SD_ref);
        end;
        with next_SD_ref^ do begin
            pos_in_record := sizeof(data) -
                             sizeof(integer) -
                             sizeof(number) -
                             sizeof(real);
            seg_size := sizeof(real);
            seg_type := seg_real32;
            sort_direction := ascending;
            next_SD_ref := nil;
            end;
    end;
```

These SDs appear in the make_employee_file procedure of the complex sample program in Appendix E.

File Information Structure

The File Information Structure, or FIS, gives you information about the structure of an RPS file. You use the FIS in conjunction with the RPS_FILEINFO call. You specify in the FIS which file you want information about, and RPS fills the rest of the FIS with information about that file. Table 2-4 shows you the layout of the FIS.

Component	Data Type	Supplied By	Comments
filename_ref	^rps_data	user to RPS	A pointer to a file name area.
record_size	integer	RPS to user	The number of bytes in the record.
record_type	scalar (fixed_length, variable_length)	RPS to user	The setting that indicates whether the records are fixed or variable length.
data_page_size	integer	RPS to user	The number of contiguous bytes used as a unit for transferring data between memory and a disk.
num_indexes	integer	RPS to user	The number of indexes for the file.
record_count	array [0..1] of integer	RPS to user	The current number of records in the file.
status	integer	RPS to user	The status of the operation upon completion.

Table 2-4. FIS Layout

Notes on FIS Components: (in bold type)

filename_ref

You use the RPS_PTR_LOAD call to fill this component. The file name must be the name of an existing RPS file.

record_size

For variable length records, RPS returns the maximum record size, which was specified in the FAS.

record count

The record count is represented as an unsigned 32 bit integer. If you know that your file has less than 32,767 records, you only need to look at the first integer of the array.

@ Key Information Structure

The Key Information Structure, or KIS, gives you information about the overall attributes of a key definition. You use the KIS in conjunction with the RPS_KEYINFO call. You specify in the KIS which specific key definition you want information about, and RPS fills the rest of the KIS with information about that key definition. Table 2-5 shows you the layout of the KIS.

Component	Data Type	Supplied By	Comments
filename_ref	^rps_data	user to RPS	A pointer to a file name area.
key_id	integer	user to RPS	The Key Identifier for the key definition.
dupls_allowed	Boolean	RPS to user	The setting that indicates whether or not duplicate key values are allowed for this key definition.
nulls_allowed	Boolean	RPS to user	The setting that indicates whether or not a null key value is allowed for this key definition.
nulls_kept	Boolean	RPS to user	This component is not currently implemented.
index_page size	integer	RPS to user	The number of contiguous bytes used as a unit for transferring key values between memory and a disk.
num_segs	integer	RPS to user	The number of segments in the key definition.
key_count	array [0..1] of integer	RPS to user	The number of key values stored in the index.
index_depth	integer	RPS to user	The number of levels in the index.
status	integer	RPS to user	The status of the operation upon completion.

Table 2-5. KIS Layout

Notes on KIS Components: (in bold type)

filename ref

You use the RPS_PTR_LOAD call to fill this component. The file name must be the name of an existing RPS file.

key count

The key count is represented as an unsigned 32 bit integer. If you know that your file has less than 32,767 key values, you only need to look at the first integer of the array.

index depth

The index depth is the number of index levels that RPS has used to hold the key values associated with this key definition. You use this information in determining the release priority array of the Initialization Structure (described in Chapter 3).

Additional Miscellaneous Notes on the KIS: (in bold type)

Note that there are only two components of the KIS that you need to supply. They are

* a pointer to the name of the file containing the key definition of interest

* the Key Identifier of the key definition of interest within the file

Segment Information Structure

The Segment Information Structure, or SIS, gives you information about the attributes of a specific key segment. You use the SIS in conjunction with the RPS_SEGINFO call. You specify in the SIS which specific key segment you want information about and RPS fills the rest of the SIS with information about that key segment. Table 2-6 shows you the layout of the SIS.

Component	Data Type	Supplied By	Comments
filename_ref	^rps_data	user to RPS	A pointer to a file name area.
key_id	integer	user to RPS	The Key Identifier for the key definition.
seg_id	integer	user to RPS	The Segment Identifier for the key segment.
pos_in_key	integer	RPS to user	The byte position within the key value where the segment begins.
pos_in_record	integer	RPS to user	The byte position within the record where the segment begins.
seg_size	integer	RPS to user	The number of bytes in the segment.
seg_type	scalar (seg_ints16, seg_intu16, seg_ints32, seg_intu32, seg_ints64, seg_intu64, seg_real32, seg_real64, seg_string, seg_packchar, seg_packbyte, seg_unknown)	RPS to user	The data type of the segment.
sort_ direction	scalar (ascending, descending)	RPS to user	The sort direction of the segment.

status	integer	RPS to user	The status of the operation upon completion.

Table 2-6. SIS Layout

Notes on SIS Components: (in bold type)

filename ref

You use the RPS_PTR_LOAD call to fill this component. The file name must be the name of an existing RPS file.

seg type

The segment types RPS returns are the same as those you supply in the SD. Refer to the section on SDs for an explanation of the segment types.

Additional Miscellaneous Notes on the SIS: (in bold type)

Note that there are only three components of the SIS that you need to supply. They are

* a pointer to the name of the file containing the key definition of interest
* the Key Identifier of the key definition of interest within the file
* the Segment Identifier of the key segment of interest

@@ File Level Calls

The following sections describe the six procedures that you use to communicate with RPS at the file level.

@ RPS_CREATE

Purpose: (in bold type)

The RPS_CREATE call creates a file with the structure specified in the associated FAS and the related KDs and SDs.

Call Format: (in bold type)

RPS_CREATE (pointer to FAS)

Components of FAS Required for Call:	filename_ref record_size record_type data_page_size KD_ref
Components of KD Required for Call:	dupls_allowed nulls_allowed nulls_kept index_page_size next_KD_ref SD_ref
Components of SD Required for Call:	pos_in_record seg_size seg_type sort_direction next_SD_ref

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1301	Record size less than limit
-1302	Record size greater than limit
-1303	Record/page size conflict
-1304	Page size less than limit
-1305	Page size greater than limit
-1308	No pointer to KD in FAS
-1309	No pointer to SD in KD
-1310	Null use invalid
-1311	Segment position invalid

-1313 Number of key definitions greater than limit
-1314 Number of key segments greater than limit
-1315 Key/page size conflict
-1410 Number of files greater than number allowed for during initialization
n SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_CREATE: (in bold type)

When you issue an RPS_CREATE call, RPS creates a file with the characteristics and structure specified in the FAS, the KDs, and the SDs.

If you look at a directory listing of the volume containing a newly created RPS file, you'll see two SOS files listed with the RPS file name, but with different suffixes. The suffix .D is for the data file, and the suffix .I is for the index file.

The directory listing for an index file lists two different file sizes. The number in the column labeled "Size" represents where the SOS end-of-file mark is set. RPS sets this mark based on the number of indexes in the file. The real file size, that is, the amount of physical space the file actually uses, appears in the column labeled "Phys".

Additional Notes on RPS_CREATE: (in bold type)

To create a file, you must supply at least one KD and at least one SD for each KD.

WARNING: Never use the Pascal MARK and RELEASE procedures and the RPS calls in the following order:

```
MARK
first RPS call
.
.
.
RELEASE
.
.
.
last RPS call
```

If you do, Pascal will release the space taken by RPS for its internal structures.

@ RPS_DESTROY

Purpose: (in bold type)

RPS_DESTROY deletes all traces of an RPS file from the system.

Call Format: (in bold type)

RPS_DESTROY (pointer to FAS)

Components of FAS Required for Call: filename_ref

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1410	Number of files greater than number allowed for during initialization
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DESTROY: (in bold type)

When you "destroy" an RPS file, you destroy both the data file and the index file. After you've destroyed an RPS file, it is completely gone from the system and cannot be used again.

@ RPS_ERASE

Purpose: (in bold type)

The RPS_ERASE call deletes all records from an RPS file, as well as all the key values in the associated indexes.

Call Format: (in bold type)

RPS_ERASE (pointer to FAS)

Components of FAS Required for Call: filename_ref

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1410	Number of files greater than number allowed for during initialization
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_ERASE: (in bold type)

When you "erase" an RPS file, you erase the contents of both the data file and the index file. The data file contains no data, but still retains the characteristics specified in the FAS when the file was created. Likewise, the index file contains no key values, but still has the characteristics specified in the KDs and SDs when the file was created.

@ RPS_FILEINFO

Purpose: (in bold type)

The RPS_FILEINFO call transfers information to you about an RPS file. The call uses the FIS to return the file information.

Call Format: (in bold type)

RPS_FILEINFO (pointer to FIS)

Components of FIS Required for Call: filename_ref

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1405	Corrupt data
-1406	Corrupt index
-1410	Number of files greater than number allowed for during initialization
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_FILEINFO: (in bold type)

When you issue an RPS_FILEINFO call, RPS fills the FIS with information about the file you're interested in. The only FIS component you need to supply is a pointer to the file-name area.

@ RPS_KEYINFO

Purpose: (in bold type)

The RPS_KEYINFO call transfers information to you about a key definition for an RPS file. The call uses the KIS to return the key definition information.

Call Format: (in bold type)

RPS_KEYINFO (pointer to KIS)

Components of KIS Required for Call:	filename_ref key_id

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1405	Corrupt data
-1406	Corrupt index
-1407	Key Identifier invalid
-1410	Number of files greater than number allowed for during initialization
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_KEYINFO: (in bold type)

When you issue an RPS_KEYINFO call, RPS fills the KIS with information about the key definition you're interested in. The only KIS components you need to supply are a pointer to the file-name area and the Key Identifier of the key definition.

@ RPS_SEGINFO

Purpose: (in bold type)

The RPS_SEGINFO call transfers information to you about a key segment for an RPS file. The call uses the SIS to return the key segment information.

Call Format: (in bold type)

RPS_SEGINFO (pointer to SIS)

Components of SIS Required for Call:
- filename_ref
- key_id
- seg_id

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1405	Corrupt data
-1406	Corrupt index
-1407	Key Identifier invalid
-1408	Segment Identifier invalid
-1410	Number of files greater than number allowed for during initialization
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_SEGINFO: (in bold type)

When you issue an RPS_SEGINFO call, RPS fills the SIS with information about the key segment you're interested in. The only SIS components you need to supply are a pointer to the file-name area, the Key Identifier of the key definition, and the Segment Identifier of the key segment.

@@@ Record Level Processing

RPS provides seven program calls to be used at the record level. The calls are listed below, along with brief descriptions of what they do.

RPS_OPENSCAN	initiates a scan on an RPS file
RPS_CLOSESCAN	ends a scan on an RPS file
RPS_FETCH	retrieves a record from an RPS file
RPS_POSITION	retrieves a key value from an index
RPS_INSERT	adds a record to an RPS file
RPS_UPDATE	rewrites a record in an RPS file
RPS_DELETE	removes a record from an RPS file

Every call contains a parameter which is a pointer to the access structure RPS uses for record level processing. You allocate and maintain the access structure in your program.

@@ Building the Scan Access Structure

RPS uses only one access structure at the record level, the Scan Access Structure, or SAS. This structure contains information relevant to the execution of a record operation. The layout of the SAS is shown in Table 2-7.

Component	Data Type	Supplied By	Comments
scan_id	integer	RPS to user	The Scan Identifier that identifies this scan. You must set this to -1 before opening a scan.
filename_ref	^rps_data	user to RPS	A pointer to a file name area.
file_use	scalar (shared, exclusive)	user to RPS	The type of use for this file for this scan.
process_ required	scalar (read_only, modify)	user to RPS	The type of processing to be performed on this file for this scan.
access_ mode	scalar (first, last, direct, current, next, prior)	user to RPS	The access mode for the scan.

record_ locking	Boolean	user to RPS	The setting that indicates whether or not RPS is to lock the record currently accessed with this scan.
key_id	integer	user to RPS	The Key Identifier that identifies which index of key values is to be used for the scan.
key_area_ref	^rps_data	user to RPS	The pointer to an area that will contain a key value.
record_ area_ref	^rps_data	user to RPS	The pointer to an area that will contain a data record.
status	integer	RPS to user	The status of the operation upon completion.

Table 2-7. SAS Layout

Notes on SAS Components: (in bold type)

scan id

RPS uses the scan_id to identify the scan in record operations. Before you open a scan for the first time, you must set the scan_id to -1 . This prevents you from opening a second scan using the same SAS while the first scan is still open. When you close the scan, RPS resets the scan_id to -1 .

shade WARNING: You must never change the Scan Identifier while a scan is open.

filename ref

You use the RPS_PTR_LOAD call to fill this component. The file name must be the name of an existing RPS file. RPS uses this component only when a scan is opened.

file use

Shared file use means that any number of scans can be open on the file. Exclusive file use means that only one scan, the scan using the SAS that claims exclusive use of the file, can be open on the file.

You might want to use this component to see if any other scans are open on the file.

RPS uses this component only when a scan is opened.

process required

Read_only processing means that records and key values can only be read from the file. Modify processing means that records can be read, changed, added, and deleted.

RPS uses this component only when a scan is opened.

record locking

You turn record locking on and off with this component. You can change the setting at any time during a scan.

Record locking provides concurrency control to any user who wishes it. If you are not concerned with concurrency control, you should set this component to false and not give it another thought.

With record locking, RPS can prevent a record from being accessed by more than one scan at the same instant in time. It works as follows:

When one scan tries to access a record, via an RPS_FETCH or RPS_POSITION call, RPS first checks to see if that record is locked.

If the record is locked, RPS looks at the record_locking component of the SAS of the scan attempting to retrieve the record.

If record_locking is set to true for this scan, access is not allowed. RPS returns no information, leaves the CROS where it was, and issues a status return indicating that the record is locked.

If record_locking is set to false for this scan, access is allowed. RPS retrieves the record and positions the CROS on the record; RPS also issues a status return informing you that the record is in use.

If the record is not locked, RPS allows access and positions the CROS on the record. Then RPS looks at the record_locking component of the SAS of the scan attempting to retrieve the record.

If record_locking is set to true, RPS then locks the record.

If record_locking is set to false, nothing else happens.

When one scan tries to modify a record, via an RPS_UPDATE or RPS_DELETE call, RPS first checks to see if another scan has locked that record. If the record is locked, the operation fails.

For insert operations, RPS executes the operations as usual and then checks the record_locking component of the SAS.

If record_locking is set to true, RPS then locks the record just added.

If record_locking is set to false, nothing else happens.

key id

Each Key Identifier corresponds to one key definition and its associated index. RPS assigns a Key Identifier to each key definition when a file is created. The numbers are assigned sequentially, beginning with 0 , according to the order of the key definitions within the linked list of KDs.

A scan has only one index associated with it.

RPS uses this component only when a scan is opened.

key area ref

You use the RPS_PTR_LOAD call to fill this component. The size of the key area equals the sum of the sizes of all key segments in the key definition.

record area ref

You use the RPS_PTR_LOAD call to fill this component. The size of the record area must be at least as large as the size indicated in the record_size component of the FAS.

Additional Miscellaneous Notes on the SAS: (in bold type)

You must exercise caution when allocating space for the record and key areas to ensure that the space is at least as large as that indicated to RPS.

SAS Example: (in bold type)

```
new(birth_SAS_ref);
with birth_SAS_ref^ do begin
    filename_ref := RPS_PTR_LOAD(name_of_file);
    file_use := shared;
    process_required := read_only;
    access_mode := direct;
    record_locking := false;
    key_id := 2;
    key_area_ref := RPS_PTR_LOAD(birth_key);
    record_area_ref := RPS_PTR_LOAD(rec)
    end;
```

This SAS appears in the list_birthdays procedure of the complex sample program in Appendix E.

@@ Record Level Calls

The following sections describe the seven program calls that you use to communicate with RPS at the record level.

@ RPS_OPENSCAN

Purpose: (in bold type)

The RPS_OPENSCAN call begins a scan on an RPS file. The call uses the SAS to communicate relevant information to RPS.

Call Format: (in bold type)

RPS_OPENSCAN (pointer to SAS)

Components of SAS Required for Call:

scan_id
filename_ref
file_use
process_required
access_mode
key_id

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1401	File not RPS file
-1403	Data file missing
-1404	Index file missing
-1405	Corrupt data
-1406	Corrupt index
-1407	Key Identifier invalid
-1409	Scan Identifier invalid
-1410	Number of files greater than maximum allowed for during initialization
-1412	Number of scans greater than maximum allowed for during initialization
-1506	File in use
-1507	File locked
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_OPENSCAN: (in bold type)

Besides identifying the file to be scanned, the RPS_OPENSCAN call performs five other functions.

* It assigns a unique Scan Identifier to the scan.

* It establishes whether this scan is to have shared or exclusive use of the file being scanned.

* It establishes the kind of processing to be used for the scan.

* It establishes which index of key values RPS is to use for the scan.

* It positions the CROS. If the access mode is First, Next, Current, or Direct, the pointer is positioned between the start-of-file and the first record in the file. If the access mode is Last or Prior, the pointer is positioned between the last record and the end-of-file.

To open a scan, RPS requires that the Scan Identifier in the SAS be set to -1 . This prevents you from opening another scan with the same SAS while the first scan is still open. When RPS opens a scan, it assigns a new unique value for the Scan Identifier. Once the Scan Identifier for a scan is set, it must never be changed; this is because the Scan Identifier is the SAS component that uniquely identifies the scan. RPS resets the Scan Identifier to -1 when the scan is closed.

Additional Notes on RPS_OPENSCAN: (in bold type)

WARNING: Never use the Pascal MARK and RELEASE procedures and the RPS calls in the following order:

```
MARK
first RPS call
.
.
.
RELEASE
.
.
.
last RPS call
```

If you do, Pascal will release the space taken by RPS for its internal structures.

@ RPS_CLOSESCAN

Purpose: (in bold type)

RPS_CLOSESCAN terminates a scan on an RPS file.

Call Format: (in bold type)

RPS_CLOSESCAN (pointer to SAS)

Components of SAS Required for Call: scan_id

Possible Status Returns:

0	Successful execution
-1405	Corrupt data
-1406	Corrupt index
-1409	Scan Identifier invalid
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_CLOSESCAN: (in bold type)

After a scan is closed, the Scan Identifier that was associated with the scan is no longer valid; RPS resets the Scan Identifier to -1 .

If an error occurs while a scan is being closed, RPS still attempts to close all files and free all resources associated with the scan.

@ RPS_FETCH

Purpose: (in bold type)

RPS_FETCH retrieves the record identified by information in the SAS and positions the CROS in the file.

Call Format: (in bold type)

RPS_FETCH (pointer to SAS)

Components of SAS Required for Call:
- scan_id
- access_mode
- record_locking
- key_area_ref
- record_area_ref

Possible Status Returns:

1003	CROS positioned at start-of-file (Prior mode only)
1002	CROS positioned at end-of-file (Next mode only)
1001	Record not found (Direct or Current mode only)
0	Successful execution
-1102	No buffer available
-1405	Corrupt data
-1406	Corrupt index
-1409	Scan Identifier invalid
-1502	Record in use by another scan
-1503	Record locked by another scan
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_FETCH: (in bold type)

The specific record retrieved by an RPS_FETCH call is determined by the access_mode component of the SAS, and, if the access mode is Direct, by the key value obtained via the key_area_ref component of the SAS.

The access modes that you use to get records are as follows:

First	Current
Last	Next
Direct	Prior

Chapter 1 contains descriptions of how these modes work.

The record retrieved is transferred to the area indicated by the record_area_ref component of the SAS. The CROS is repositioned at the record selected by the RPS_FETCH.

If the key value specified for an RPS_FETCH using Direct access mode does not exist, no data is returned to the record area; the CROS is positioned between the records whose key values bracket the specified value; and a status code is returned indicating that there is no such record.

Record Locking Consequences. If the record you are trying to retrieve is "locked", that is, if another scan is accessing that record and the record_locking component of its SAS is set to true, RPS takes one of two actions, depending on whether the record_locking component of your scan is set to true or false.

> If your scan's record locking is true, you cannnot access the record. RPS returns the status return -1503, which tells you that the record is locked. You will not be able to access the record until the other scan has unlocked it. The position of the CROS is unchanged.
>
> If your scan's record locking is false, RPS fetches the record, but returns the status return -1502, which tells you that the record is in use. RPS repositions the CROS to point to the chosen record.

After any successful execution of an RPS_FETCH call, RPS checks the record_locking component of the SAS. If it's set to true, RPS locks the record. If it's set to false, nothing else happens.

Additional Notes on RPS_FETCH: (in bold type)

CROS Positioning. With the access_mode component of the SAS set to First or Last, the RPS_FETCH call does not return any data; it simply repositions the CROS to the start or end of the file, respectively. The status code returned is 0 . RPS returns status code for start-of-file (1003) or end-of-file (1002) only when the access mode is Prior or Next, respectively.

Changing Access Modes. While a scan of a file is open, you can change the access_mode component of the SAS at any time. For example, you might want to execute an RPS_FETCH using Direct access mode for a specific key value. Then, after the RPS_FETCH is executed successfully, you can set the access_mode component to Next. Subsequent RPS_FETCH calls would return records in a sequential order determined by the sort order of the records, based on the index being used for the scan.

@ RPS_POSITION

Purpose: (in bold type)

RPS_POSITION retrieves the key value identified by information in the SAS and positions the CROS in the file.

Call Format: (in bold type)

RPS_POSITION (pointer to SAS)

Components of SAS Required for Call: scan_id
access_mode
record_locking
key_area_ref

Possible Status Returns:

1003	CROS positioned at start-of-file (Prior mode only)
1002	CROS positioned at end-of-file (Next mode only)
1001	Record not found (Direct or Current mode only)
0	Successful execution
-1102	No buffer available
-1405	Corrupt data
-1406	Corrupt index
-1409	Scan Identifier invalid
-1502	Record in use by another scan
-1503	Record locked by another scan
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_POSITION: (in bold type)

The key value retrieved by an RPS_POSITION call is transferred to the area indicated by the key_area_ref component of the SAS. The CROS is repositioned at the record containing the specified key value.

The access modes that you use to get key values are as follows:

First	Current
Last	Next
Direct	Prior

Chapter 1 and the next section, Additional Notes on RPS_POSITION, contain descriptions of how these modes work.

The main functions of the RPS_POSITION call are to find a specific key value without having to retrieve entire records, and to move the CROS.

Record Locking Consequences. If the record whose key value you are trying to retrieve is "locked", that is, if another scan is accessing that record and the record_locking component of its SAS is set to true, RPS takes one of two actions, depending on whether the record_locking component of your scan is set to true or false.

> If your scan's record locking is false, RPS returns the key value, but returns the status return -1502, which tells you that the record is in use. RPS repositions the CROS to point to the record with the chosen key value.
>
> If your scan's record locking is true, you cannnot access the record's key value. RPS returns the status return -1503, which tells you that the record is locked. You will not be able to access the value until the other scan has unlocked the record. The position of the CROS is unchanged.

After any successful execution of an RPS_POSITION call, RPS checks the record_locking component of the SAS. If it's set to true, RPS locks the record. If it's set to false, nothing else happens.

Additional Notes on RPS_POSITION: (in bold type)

The specific key value retrieved by an RPS_POSITION call is determined by the access_mode component of the SAS.

CROS Positioning with Direct Mode. If the access mode is Direct, the key value specified by the key_area_ref component of the SAS is used to retrieve the key value. Thus, since the key value is already known, using the RPS_POSITION call with Direct access mode is just a way of positioning the CROS without returning the data record.

If the key value specified for an RPS_POSITION call using Direct access mode does not exist, no data is returned to the key area; however, the CROS is positioned between the records whose key values bracket the specified value; and a status code is returned indicating that there is no such record. At this point, you could set the access mode to Next and retrieve the next key value.

CROS Positioning with Other Modes. With the access_mode component of the SAS set to First or Last, the RPS_POSITION call does not return any key value; it simply repositions the CROS to the start or end of the file, respectively. The status code returned is 0 . RPS returns status code for start-of-file (1003) or end-of-file (1002) only when the access mode is Prior or Next, respectively.

@ RPS_INSERT

Purpose: (in bold type)

RPS_INSERT adds a record to an RPS file and positions the CROS on that record.

Call Format: (in bold type)

RPS_INSERT (pointer to SAS)

Components of SAS Required for Call: scan_id
record_area_ref

Possible Status Returns:

0 Successful execution
-1102 No buffer available
-1301 Record size less than limit
-1302 Record size greater than limit
-1405 Corrupt data
-1406 Corrupt index
-1409 Scan Identifier invalid
-1504 Processing attempted not allowed
-1505 No path to record
-1550 to -1557 No duplicate key values allowed in index. The rightmost digit of the status return (0 through 7) tells the index in which you tried to put the duplicate key value.
-1560 to -1567 No null key values allowed in index. The rightmost digit of the status return (0 through 7) tells the index for which there was a null value.
-1570 to -1577 Index too large. The rightmost digit of the status return (0 through 7) tells the index that was too large.
-1580 to -1587 Index level too deep. The rightmost digit of the status return (0 through 7) tells the index for which there were too many key values.
n SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_INSERT: (in bold type)

The RPS_INSERT call adds the record indicated by the record_area_ref component of the SAS to the file associated with the scan_id. When you add a record to a file, RPS inserts all key values for that record into the respective indexes. After the record is added, the CROS is updated to point to the new record. The position of the CROS before the call is immaterial.

When you insert a record, RPS puts it in any available space within the file. RPS extends a file only when there is no more free space available in the file. RPS keeps track of the space made available in a file when a record is deleted and uses that space for new records.

Restrictions on Duplicate and Null Key Values. If you try to add a record that violates the restrictions on duplicate key values or null key values, RPS does not add the record to the file; the position of the CROS is unchanged.

Restriction on Variable Length Records. If you're using variable length records and you try to add a record that is smaller than the minimum record size for the file, RPS does not add the record to the file. This is because it cannot extract all key values needed for the associated indexes.

Likewise, if the length of the record you're trying to add exceeds the maximum size allowed, RPS does not add the record.

In both cases, the position of the CROS is unchanged.

Additional Notes on RPS_INSERT: (in bold type)

To add a record to an RPS file, you must specify modify for the process_required component of the SAS when the scan is opened.

If you insert a record containing a null value for a key value, the index for that key definition will have no reference to the added record.

@ RPS_UPDATE

Purpose: (in bold type)

RPS_UPDATE rewrites a record in an RPS file and leaves the CROS positioned on that record.

Call Format: (in bold type)

RPS_UPDATE (pointer to SAS)

Components of SAS Required for Call: scan_id
record_area_ref

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1301	Record size less than limit
-1302	Record size greater than limit
-1405	Corrupt data
-1406	Corrupt index
-1409	Scan Identifier invalid
-1501	CROS position invalid
-1504	Processing attempted not allowed
-1505	No path to record
-1550 to -1557	No duplicate key values allowed in index. The rightmost digit of the status return (0 through 7) tells the index in which you tried to put the duplicate key value.
-1560 to -1567	No null key values allowed in index. The rightmost digit of the status return (0 through 7) tells the index for which there was a null value.
-1570 to -1577	Index too large. The rightmost digit of the status return (0 through 7) tells the index that was too large.
-1580 to -1587	Index level too deep. The rightmost digit of the status return (0 through 7) tells the index for which there were too many key values.
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_UPDATE: (in bold type)

The RPS_UPDATE call transfers data from the area indicated by the record_area_ref to the record at which the CROS is positioned. If the CROS is positioned between two records, or between the start-of-file and the first record or the end-of-file and the last record, no update occurs. RPS returns a status code indicating that the position of the CROS is invalid.

Restrictions on Duplicate and Null Key Values. If you try to update a record in a way that violates the restrictions on duplicate key values or null key values, RPS does not update the record in the file; the position of the CROS is unchanged.

Restriction on Variable Length Records. If you're using variable length records and you try to update a record so that it becomes smaller than the minimum record size for the file, RPS does not update the record in the file. This is because it cannot extract all key values needed for the associated indexes.

Likewise, if the length of the record you're trying to update becomes greater than the maximum size allowed, RPS does not update the record.

In both cases, the position of the CROS is unchanged.

Additional Notes on RPS_UPDATE: (in bold type)

To update a record, the process_required component of the SAS must specify modify when the scan is opened.

@ RPS_DELETE

Purpose: (in bold type)

RPS_DELETE removes a record from an RPS file and positions the CROS between the records that bracket the deleted record.

Call Format: (in bold type)

RPS_DELETE (pointer to SAS)

Components of SAS Required for Call: scan_id

Possible Status Returns:

0	Successful execution
-1102	No buffer available
-1405	Corrupt data
-1406	Corrupt index
-1409	Scan Identifier invalid
-1501	CROS position invalid
-1504	Processing attempted not allowed
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DELETE: (in bold type)

RPS_DELETE removes the record at which the CROS is positioned. After a record is deleted, the CROS is updated to point between the records that bracket the deleted record.

If you issue an RPS_DELETE call and the CROS is not pointing directly to a record, no deletion occurs; RPS returns an appropriate status code.

When a record is deleted, all key values associated with that record are removed from the respective indexes.

RPS keeps track of the space made available when a record is deleted. When you insert a record, RPS reuses that space before extending the file. The file is extended only when there is no more free space available.

Additional Notes on RPS_DELETE: (in bold type)

To delete a record, the process_required component of the SAS must specify modify when the scan is opened.

@@@@ Chapter 3

@@@ Guidelines for Memory Initialization and Optimization

The primary purpose of this chapter is to tell you how to optimize the performance of RPS for your particular application. You accomplish this by giving RPS values to use in determining its memory allocation, rather than letting RPS use its default allocation.

Normally you don't need to be concerned about optimizing performance until after your application is running properly. However, if the application requires any of the following characteristics, you must change the default memory allocation before the program will run.

* Any key value size is larger than 20 bytes.
* Any record size is larger than 100 bytes.
* You want to scan more than one file at a time.
* You want to have more than two scans open at a time.
* Any page size is larger than 512 bytes.

In addition to making sure that RPS allocates enough memory to handle your application, you use the optimization techniques to adjust the space/performance ratio for your particular application so that the application performs in the most efficient way possible on the Apple III.

The primary way to optimize an application is to give RPS certain information about the application, which RPS in turn uses in allocating space for its internal use and for managing buffers for paging. You include the RPS_INITIALIZE call and an associated access structure, the Initialization Structure, in your program to supply the information RPS needs.

@@@ How This Chapter Is Organized

This chapter has four main parts:

The first part introduces you to the Initialization Structure (IS). It tells you what initialization is, it gives you some key points about the IS, and it shows you the complete layout of the IS.

The second part contains the details of how to specify the IS, component by component.

The third part tells you how to specify the RPS_INITIALIZE call.

The fourth part summarizes all the information in the first three parts.

@@@ Introducing the IS

The IS has thirteen components that you need to specify, plus the same status return component that the other access structures have. When you issue the RPS_INITIALIZE call, RPS initializes its memory area, using the values in the IS to determine how large an area is needed.

@@ What Does Initialization Mean?

RPS "initializes" its memory when you issue the first RPS call in your program. To initialize memory means that RPS first determines how much space it needs to interact with your program and then sets up that space.

Figure 3-1 shows how RPS manages its memory area.

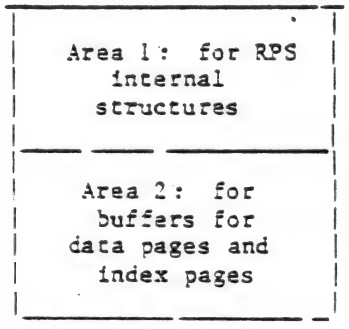


Figure 3-1. RPS Memory Area

This shows you that RPS actually divides its memory area into two separate parts. Area 1 contains the internal RPS structures that describe the files, indexes, and scans being used in an application. The contents of this area change as you change the files being scanned and the indexes being used.

Any part of Area 1 can be used for any of the internal RPS structures. RPS carefully manages the locations of the structures. As structures are no longer needed during the course of a program's execution, RPS frees the area holding the structures for other use.

Area 2 contains the buffers that hold the data pages and index pages. RPS determines the exact number of buffers from the values you supply in the IS.

RPS determines the size of these memory areas in one of two ways:

If you include the RPS_INITIALIZE call, RPS uses the values that you specify in the IS to calculate the size.

or

If you don't include the RPS_INITIALIZE call, RPS uses its own values. Initialization occurs when you issue the first RPS call in the program.

@@ Some Key Points About the IS

Key Point #1. It's up to you to choose whether or not you use the IS. However, you must use it if:

* Any key value size is larger than 20 bytes.
* Any record size is larger than 100 bytes.
* You want to scan more than one file at a time.
* You want to have more than two scans open at a time.
* Any page size is larger than 512 bytes.

If you choose not to use the IS, RPS supplies its own values for all the components. If you choose to use the IS, you must supply values for all the components. In other words, it's an all or nothing situation.

Key Point #2. If you decide to use the IS, you must build the structure and issue the RPS_INITIALIZE call before issuing any other RPS call.

Key Point #3. You allocate and maintain the IS as part of your program, just as you do the access structures described in Chapter 2.

@@ The Layout of the IS

Table 3-1 shows you the complete layout of the IS. The next section contains the details on how to build the IS.

Component	Data Type	Supplied By	Comments
key_size	integer	user to RPS	The number of bytes of the longest key value to be used.
record_size	integer	user to RPS	The number of bytes of the longest record to be updated.
num_files	integer	user to RPS	The maximum number of files to be scanned at one time.
num_scans	integer	user to RPS	The maximum number of scans to be opened at one time.
num_indexes	integer	user to RPS	The maximum number of indexes in use at one time.
num_segs	integer	user to RPS	The maximum total number of segments in use at one time.
num_512_ buffs	integer	user to RPS	The number of 512 byte buffers to be allocated for paging.
num_1024_ buffs	integer	user to RPS	The number of 1024 byte buffers to be allocated for paging.
num_1536_ buffs	integer	user to RPS	The number of 1536 byte buffers to be allocated for paging.
num_2048_ buffs	integer	user to RPS	The number of 2048 byte buffers to be allocated for paging.

rel_priority	array [0..11] of integer	user to RPS	An array of priorities which will determine the order in which pages are released from main memory.
write_ threshold	integer	user to RPS	The maximum number of records RPS may change in data buffers before writing the buffers out to the file.
extend_size	integer	user to RPS	The number of pages to be added to a file when all the available space in the file has been used.
status	integer	RPS to user	The status of the operation upon completion.

Table 3-1. IS Layout

@@@ Building the IS

Keeping in mind the categories just described, the components of the IS are:

Space Allocation Components	Buffer Management Components
key_size	num_512_buffs
record_size	num_1024_buffs
num_files	num_1536_buffs
num_scans	num_2048_buffs
num_indexes	rel_priority
num_segs	write_threshold
	extend_size

@@ Space Allocation Components

While a program using RPS is executing, RPS allocates and releases chunks of its memory area to hold the internal structures it needs. This process corresponds to the process of opening and closing scans; because, for each scan, RPS needs to build and maintain certain file, index, and scan information in its internal structures.

The next two sections present a model of space allocation and give background information on how RPS uses its space. The six sections following that describe the six space allocation components of the IS. Each section tells what RPS does with the component value you supply and how to determine the correct value for your application.

Time Line Model. Think of the execution of an application as a sequence of interactions with RPS along a time line. The amount of memory RPS requires can vary from one instant to another, from one interaction to another, as shown in Figure 3-2.

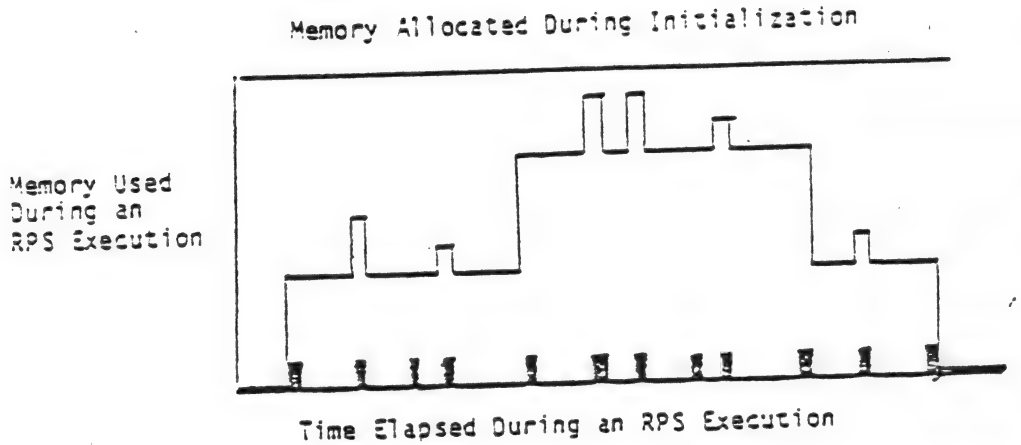


Figure 3-2. Time Line Model

Each ▌ along the bottom line marks the period in time when an RPS operation occurs.

The purpose of specifying the space allocation components is to set aside enough memory so that RPS will have room for the maximum space needed for internal structures.

You don't actually compute the allocation yourself; RPS uses the numbers you supply in its own calculations. The result is the maximum amount of memory RPS must have available for the application.

@ Background Information on Space Allocation

RPS Files. For every RPS file that you define, RPS creates two SOS files, an index file and a data file. If you look at a directory listing of the volume containing your RPS file, you'll see two files listed with the RPS file name, but differentiated by the suffixes. Figure 3-3 shows the relationship between the RPS file and the SOS files.

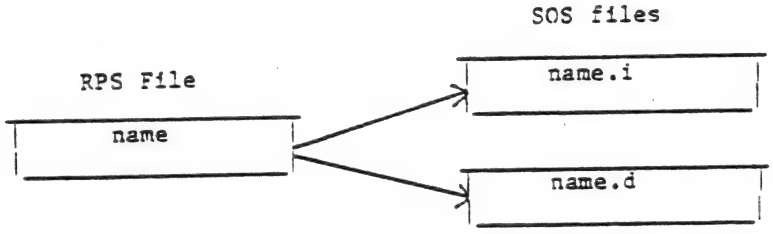


Figure 3-3. Relationship Between RPS and SOS Files

The data file contains the data records, and the index file contains all the indexes for the data file.

Internal Structures. RPS uses three types of internal structures when processing an RPS file:

the File Descriptor
the Scan Descriptor
the Index Descriptor

These are the structures for which RPS allocates Area 1 of memory, as described above. The number of each kind of descriptor that RPS keeps in memory at any one time depends solely on the kind of actions occurring in the program. The varied processing that occurs in a program results in the dynamic allocating and releasing of space for these structures.

A File Descriptor contains information that RPS uses for its processing of the file. A File Descriptor is 196 bytes long. RPS builds one File Descriptor for a file the first time you open a scan on that file. If you later open more scans on the same file, RPS uses the same File Descriptor. When all scans of the file are closed, RPS releases the area reserved by the File Descriptor.

A Scan Descriptor contains information about a specific scan. A Scan Descriptor is 26 bytes long. RPS builds one Scan Descriptor for every scan you open. When a scan is closed, RPS releases the area held by the Scan Descriptor.

An Index Descriptor contains information about a specific index. An Index Descriptor is 44 bytes long. RPS builds an Index Descriptor for each index that might have to be used for a scan. When a scan is closed, RPS checks to see if the associated index is being used by any other scan. If it isn't, RPS releases the space reserved by the Index Descriptor.

@ key_size

The key_size component of the IS is the number of bytes of the longest key value size used at any instant during the program's execution.

Reason for Supplying: (in bold type)

RPS uses the number you supply for this component to determine how much space to set aside for temporarily storing key values. RPS stores key values as part of the process of setting the CROS position.

Amount of Space Allocated: (in bold type)

RPS allocates four times the number of bytes you specify.

Default Supplied by RPS: (in bold type) 20 bytes

How to Determine This Component: (in bold type)

Compute the key size for each key definition being used for each scan; then take the maximum of those figures and add 2 to that number. To compute the key size for a key definition, add together the sizes of all the key segments in the definition.

The value for key_size must be greater than or equal to 1 and less than or equal to 2048 .

Example: (in bold type)

In this example, 3 files are being scanned simultaneously.

File 1 has 4 key definitions.
Keydef 1 is 6 bytes.
Keydef 2 is 6 bytes.
Keydef 3 is 3 bytes.
Keydef 4 is 10 bytes.

File 2 has 2 key definitions.
Keydef 1 is 12 bytes.
Keydef 2 is 6 bytes.

File 3 has 3 key definitions.
Keydef 1 is 8 bytes.
Keydef 2 is 2 bytes.
Keydef 3 is 14 bytes.

Given these key sizes, consider this possible sequence of events:

1. Scan 1 is opened on File 1 using Keydef 1 (6 bytes).
2. Scan 2 is opened on File 2 using Keydef 1 (12 bytes).
3. Scan 3 is opened on File 3 using Keydef 2 (2 bytes).
4. Scan 4 is opened on File 1 using Keydef 4 (10 bytes).

In this case, the number you would specify for key_size is 14. This is because the longest key value size being used for a scan is 12 bytes long and you add 2 to that number.

@ record_size

The record_size component of the IS is the number of bytes of the longest record updated.

Reason for Supplying: (in bold type)

RPS uses the number you supply for this component to determine how much space to set aside for temporarily storing the record during an update.

Amount of Space Allocated: (in bold type)

RPS allocates exactly the number of bytes you specify.

Default Supplied by RPS: (in bold type) 100 bytes

How to Determine This Component: (in bold type)

Consider the sizes of all the records to be updated, both before and after the update; then use the maximum of those numbers.

The value for record_size must be greater than or equal to 1 and less than or equal to 2048 .

@ num_files

The num_files component of the IS is the maximum number of different RPS files to be scanned at one time.

Reason for Supplying: (in bold type)

RPS uses the number you supply to figure out how much area to set aside for its internal File Descriptors and for SOS I/O buffers.

Amount of Space Allocated: (in bold type)

$$S_{file} = n_{file} * (196 + 2 * 1024)$$

where S_{file} is the total number of bytes to be allocated

n_{file} is the number of files you specify

Default Supplied by RPS: (in bold type) 1 file

How to Determine This Component: (in bold type)

Count the number of different files being scanned simultaneously at each point in your application and use the maximum number.

The value for num_files must be greater than or equal to 1 .

Example: (in bold type)

Consider an application consisting of this sequence of events.

	Scan Condition	Number of Different Files Scanned Simultaneously
1.	two scans open on two different files, File 1 and File 2	2
2.	a second scan open on File 2	2
3.	a scan open on File 3	3
4.	one scan on File 2 closed	3
5.	the scan on File 1 closed	2
6.	scans on File 2 and File 3 closed	0

The number you would specify for num_files is 3, because that is the maximum number of files scanned simultaneously.

@ num_scans

The num_scans component of the IS is the maximum number of scans open at one time.

Reason for Supplying: (in bold type)

RPS uses the number you supply to figure out how much area to set aside for its internal Scan Descriptors.

Amount of Space Allocated: (in bold type)

$$S_{scan} = (n_{scan} + 1)(n_{ksize} + 26)$$

where S_{scan} is the total number of bytes to be allocated

n_{scan} is the number of scans you specify

n_{ksize} is the number you specified for the key_size component

Default Supplied by RPS: (in bold type) 2 scans

How to Determine This Component: (in bold type)

Count the number of scans open simultaneously at every instant in your application and use the maximum number.

The value for num_scans must be greater than or equal to 1 .

Example: (in bold type)

Consider an application consisting of this sequence of events.

	Scan Condition	Number of Scans Open Simultaneously
1.	two scans open on two different files, File 1 and File 2	2
2.	a second scan open on File 2	3
3.	a scan open on File 3	4
4.	one scan on File 2 closed	3
5.	the scan on File 1 closed	2
6.	scans on File 2 and File 3 closed	0

The number you would specify for num_scans is 4, because that is the maximum number of scans in progress at any one time.

@ num_indexes

The num_indexes component of the IS is the maximum number of indexes in use at one time.

Reason for Supplying: (in bold type)

RPS uses the number you supply to figure out how much area to set aside for its internal Index Descriptors.

Amount of Space Allocated: (in bold type)

$$S_{index} = n_{index} * 44$$

where S_{index} is the total number of bytes to be allocated
n_{index} is the number of indexes you specify

Default Supplied by RPS: (in bold type) 4 indexes

How to Determine This Component: (in bold type)

Consider these important facts before counting the number of indexes.

* When you open a scan for read-only processing, RPS builds an Index Descriptor only for the index associated with the scan.

* When you open a scan for modify processing, however, RPS builds an Index Descriptor for every index defined for the file being scanned. This is because RPS assumes that any changes made to the data file will affect all indexes as well. For example, if you insert a record, RPS must add the new key values to the respective indexes. Thus, RPS must access all the indexes, not just the one being used for the scan.

* Once RPS builds an Index Descriptor for a given index, RPS will use that Index Descriptor for any other scan that accesses the same index.

Keeping these facts in mind, examine your application. For every point in time, count the number of Index Descriptors RPS needs to use and take the maximum of those numbers.

The value for num_indexes must be greater than or equal to 1 and less than or equal to the result of the calculation (num_files * 8).

Example: (in bold type)

Given that file 1 has 5 indexes and file 2 has 3 indexes, consider this sequence of events.

	Scan Condition	Indexes in Use for File 1 (5 max)	Indexes in Use for File 2 (3 max)	Total Indexes in Use (8 max)
1.	one scan open for read-only on File 1 using Keydef 1	1	0	1
2.	one scan open for modify on File 2	1	3	4
3.	a scan open for modify on File 1 using Keydef 3	5	3	8
4.	a second scan open for read-only on File 1 using Keydef 2	5	3	8
5.	the modify scan on File 1 closed	2	3	5

The number you would specify for num_indexes is 8 because that is the maximum number of indexes in use at any one time.

Notice that closing the modify scan on File 1 causes the number of indexes in use for File 1 to change from 5 to 2. This is because RPS releases the Index Descriptors of all indexes except the ones being used for the read-only scans.

@ num_segs

The num_segs component of the IS is the maximum total number of key segments to be used at one time.

Reason for Supplying: (in bold type)

RPS uses the number you supply to figure out how much area to set aside for its internal segment description structures.

Amount of Space Allocated: (in bold type)

$$S_{seg} = n_{seg} * 16$$

where S_{seg} is the total number of bytes to be allocated

n_{seg} is the number of segments you specify

Default Supplied by RPS: (in bold type) 16 segments

How to Determine This Component: (in bold type)

First, count the number of key definitions of every file accessed in your program. Then, at every time point in the program, take the sum of the total number of key segments for all the files being scanned. Use the maximum of those numbers. Note that you count the key segments of all the key definitions for the files being scanned, not just the key definition used for the scan.

The value for num_segs must be greater than or equal to 1 and less than or equal to the result of the calculation (num_files * 64).

Example: (in bold type)

Suppose you have 3 files with the following structures.

File 1 has 4 key definitions.
Keydef 1 has 1 segment.
Keydef 2 has 1 segment.
Keydef 3 has 1 segment.
Keydef 4 has 3 segments.
Total segments for File 1 = 6

File 2 has 2 key definitions.
Keydef 1 has 3 segments.
Keydef 2 has 2 segments.
Total segments for File 2 = 5

File 3 has 2 key definitions.
Keydef 1 has 1 segment.
Keydef 2 has 4 segments.
Total segments for File 3 = 5

If there is a point during program execution when all three files are being scanned, the number you would supply for num_segs is 16, because that is the maximum sum of key segments.

If there is a point in time during program execution when File 1 and File 2 are being scanned, and File 3 is never scanned when the other two files are being scanned, the number you would supply for num_segs is 11, because that is the maximum sum of key segments.

@ Summary of Space Allocation

RPS uses the following process to determine how much space to allocate:

* RPS takes the values you specify for the space allocation components in the IS and solves each equation for the amount of space to be allocated for that type of structure.

* RPS then adds together the results of those calculations to determine the total space to be allocated.

If you've given RPS all the right figures, the total space should be large enough to hold all the needed structures and yet small enough so that space is not wasted.

WARNING: Never use the Pascal MARK and RELEASE procedures and the RPS calls in this order:

```
MARK
first RPS call
.
.
.
RELEASE
.
.
.
last RPS call
```

If you do, Pascal will release the space taken by RPS for its internal structures.

@@ Buffer Management Components

RPS uses the buffer management components of the IS to allocate and manage space for data records and key values during program execution.

@ Number of Buffers

RPS uses buffers to hold data and index pages while they are in memory. Four components of the IS control the allocation of buffers for paging. These components are:

num_512_buffs
num_1024_buffs
num_1536_buffs
num_2048_buffs

Reason for Supplying: (in bold type)

RPS allocates the number of buffers you request in the IS. The buffers come in four sizes: 512 bytes, 1024 bytes, 1536 bytes, and 2048 bytes. Each size corresponds to one of the IS components listed above.

Amount of Space Allocated: (in bold type)

$$S_{buff} = n_{.5K}(526) + n_{1.0K}(1038) + n_{1.5K}(1550) + n_{2.0K}(2062)$$

where S_{buff} is the total number of bytes to be allocated

$n_{.5K}$ is the number of 512-byte buffers you specify

$n_{1.0K}$ is the number of 1024-byte buffers you specify

$n_{1.5K}$ is the number of 1536-byte buffers you specify

$n_{2.0K}$ is the number of 2048-byte buffers you specify

Default Supplied by RPS: (in bold type)

8 buffers of 512 bytes
0 buffers of 1024 bytes
0 buffers of 1536 bytes
0 buffers of 2048 bytes

How to Determine This Component: (in bold type)

The sizes of buffers you need to allocate depends on the sizes of the pages defined when the file was created.

Page Size Range	Buffer Size Required
14 to 512	512
513 to 1024	1024
1025 to 1536	1536
1537 to 2048	2048

You must allocate at least one buffer for each range that contains a page size defined for the file. This applies to both data and index page sizes.

In addition, if you plan to modify a file, you must allocate at least three buffers for each range that contains an index page size defined for the file.

You should experiment with different buffer sizes to improve the performance of your program. You have to choose between the improved performance of more pages in memory and the reduction of stack-heap space for your program. There are no sure-fire rules for these components, so just take your best shot.

Example: (in bold type)

Suppose your RPS file has the following page sizes:

```
512 bytes  (data page size)
600 bytes  (index page size for an index to be used for
            read-only)
400 bytes  (index page size for an index to be used for
            modify)
1325 bytes (index page size for an index to be used for
            modify)
```

For these pages sizes, you must allocate at least

```
3 buffers of 512 bytes
1 buffer of 1024 bytes
3 buffers of 1536 bytes
```

Note that the index with the 600-byte page size is to be used for read-only; therefore, it has only one buffer allocated.

@ rel_priority

The rel_priority component determines the priorities of the pages in memory.

Reason for Supplying: (in bold type)

RPS uses these priorities when deciding which page to release from main memory to make room for another page. For more reasons, refer to the next section on background information.

Default Supplied by RPS: (in bold type) 0,10,10,...,10

How to Determine This Component: (in bold type)

Background Information

Tree Structures. RPS organizes each index in an index file in a form called an n-way tree structure. Figure 3-4 shows the form of an n-way tree.

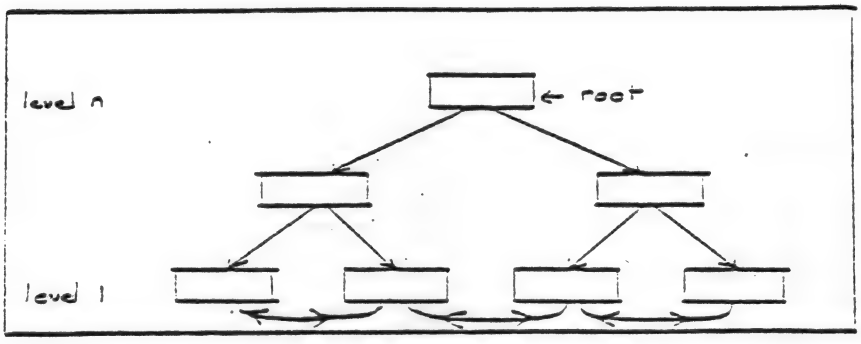


Figure 3-4. Tree Structure Diagram

Each ▭ represents a page of key values. The page at the top, which is called the root, contains pointers to pages at the next lower level. Those pages in turn have pointers to pages at the level below that. Finally, each page at the lowest level points to records in the data file. The pages at the lowest index level are linked together.

The root level is level n and the lowest index level is level 1 . RPS considers the data pages to be on level 0 .

An index can have as many as ten levels. However, most indexes that RPS builds typically have two, three, or four levels. You can find out how many levels an index has by looking at the index_depth component of the KIS associated with that index.

To find a specific key value, RPS searches down the tree using a series of compares. The search begins at the root and proceeds down the tree to the lowest level.

How RPS Fetches Data. Keeping in mind that each and every index is organized into levels of pages, now consider the way RPS fetches data, both record-type data and index-type data.

When fetching any data, RPS first looks for the data in main memory and then, if the data isn't in main memory, goes out to the disk to find it.

Because the Apple III's memory has a finite amount of space, RPS must sometimes empty out a data area presently in memory to make room for data to be brought in from disk. Fortunately, both main memory and disk files are divided into pages, which can be transferred back and forth interchangeably.

Figure 3-5 shows comparable pages in main memory and a disk file. The prime marks (') after the page indications in main memory mean that those pages are copies of the pages in the disk file.

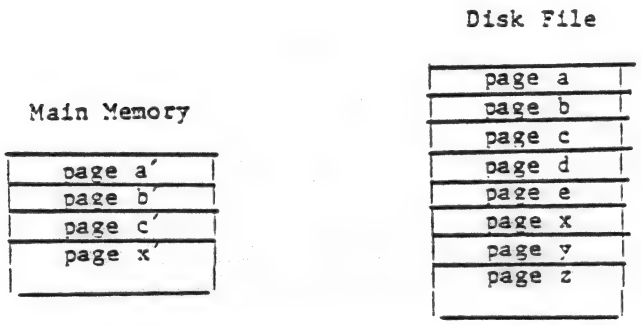


Figure 3-5. Pages of Main Memory and a Disk File

Using the picture above, suppose you need one of the key values stored in page z. RPS first looks for the value in main memory. Not finding the value, RPS then goes out to the disk, finds the value in page z, and brings that page into memory, as shown in Figure 3-6.

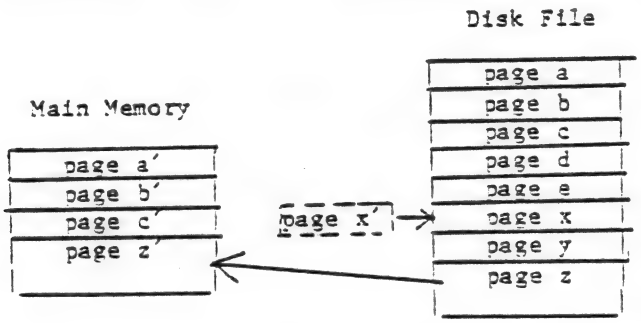


Figure 3-6. The Paging Process

Notice that RPS had to empty out one of the pages in memory to make room for the data in page z. In this case, RPS chose to release page x. The reason why RPS chose page x is that page x was the "oldest" page in memory. Page aging is explained in the next section.

Depending on the type of processing you're doing, you might be using the pages at one level of an index more frequently than the pages at another level. Thus, to improve performance, it makes sense to keep the most frequently used pages in memory.

Page Aging. To make sure that the most frequently used pages stay in memory as much as possible, RPS uses a "page aging" technique.

Page aging means that each page in memory from an index file or a data file has an age relative to the ages of other pages in memory. In deciding which page to release from memory, RPS chooses pages in the order from oldest to youngest. The older the page, the more likely it is to be released. The younger the page, the more likely it is to remain in memory.

RPS determines the relative age of a page using two pieces of information:

* the count registered on the internal RPS clock
* the release priority you assign to the level of the page

The sum of these values is the age of the page.

RPS considers the ages of all the buffers of a given size when determining which page to release. In other words, RPS looks at the 512 byte buffers as a group, the 1024 byte buffers as another group, and so on.

Purpose of Release Priorities. Because of the way RPS computes page age, the purpose of assigning release priorities is to make the pages in memory on a given index level appear younger or older to RPS than they really are. The release priority is in effect the number of extra clock ticks to be added to a page's age to artificially change the age.

The higher the release priority, the younger the pages appear, and the more likely they are to stay in memory. The lower the release priority, the older the pages appear, and the less likely they are to stay in memory.

The Release Priority Array. Using this array, you assign a priority value to each level of an index and to the data level. The value for a specific level applies to all pages in memory on that level. In addition, the entire array applies to every index used.

Release priority values can be anywhere in the range from -1 to 100. The value -1 makes a page as old as the oldest page. Thus, it will be the first to go when the time comes to release a page.

You specify the values in the release priority array in the following order:

```
data level (level 0)
lowest index level (level 1)
next lower index level (level 2)
.
.
.
root level (level n)
```

Example of How Release Priorities Work. In this example, we've given the three levels of an index the priority values 15, 10, and 5, respectively, and the data level the value 0.

Level	Priority Value	Pages on This Level
0	0	d1, d2, d3, d4
1	5	c, d, f, g
2	10	b, e
3	15	a

Now suppose that we're using this three-level index to access a file, and that we're using four buffers of memory to hold the pages from disk. To find a key value contained in page c and then get the associated record, which is contained in page d2, RPS fills the buffers as shown in Figure 3-7.

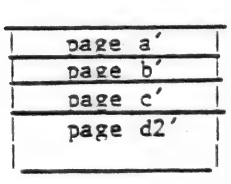


Figure 3-7. Buffers in Main Memory

For the next direct find, suppose RPS needs to bring in page e, then page f, to find the exact key value. To make room for these pages, RPS uses its internal clock and the priority scheme we've set up in deciding which pages to release from memory.

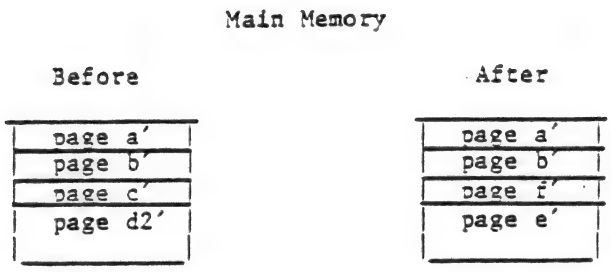


Figure 3-8. Changed Contents of Buffers

What Happened. Pages a and b stayed in memory because they have high priorities, 15 and 10, respectively. Pages c and d2, however, have lower priorities, 5 and 0, respectively; thus, RPS released these pages to make room for pages e and f.

Tips on Choosing Release Priorities

This section contains tips to help you decide on the priorities most suitable to your application.

First, use a short program containing the RPS_KEYINFO call to find out how many levels each index has. The index_depth component of the KIS tells you the depth of any index in the file.

Second, decide whether your application does more sequential processing or more random processing, in general.

Third, use this information and the tips below to set up the array of release priorities in the IS.

Tip #1: You should carefully consider the structures of all your indexes when choosing release priorities.

Reason: Release priorities apply globally to your entire RPS application. This means that the set of priorities that you establish in the IS applies to all indexes for all files used during the RPS run.

Tip #2: For applications that specialize in sequential processing, give the lowest level of the index the highest priority, and give the root level and the intermediate levels lower priorities.

Reason: When processing records sequentially, RPS uses the root level for the first record access and never uses it again. Subsequently, RPS uses the lowest index level for every record access.

Remember that the pages at the lowest index level are linked together. When RPS finishes going through all the key values in a given page at this level, it brings in the next or previous linked page of key values at that level, depending on which access mode is being used. Thus, since RPS will always use the lowest level pages, keeping those pages in memory provides the most efficient processing.

Tip #3: For applications that specialize in random processing, give the root level the highest priority and the other levels progressively lower priorities.

Reason: RPS always accesses the root level for random processing, so you want to keep that level in memory as much as possible. RPS accesses the other levels less frequently as it searches down the tree.

Summary of Key Points About Release Priorities

1. Release priorities are global for your entire RPS application. This means that the set of priorities that you establish in the IS applies to all indexes for all files used during the RPS run.

2. The higher the release priority, the younger the pages of a given index level appear, and the more likely they are to stay in memory. The lower the release priority, the older the pages appear, and the less likely they are to stay in memory.

3. RPS considers the ages of all the buffers of a given size when determining which page to release. In other words, RPS looks at the 512 byte buffers as a group, the 1024 byte buffers as another group, and so on.

@ write_threshold

The write_threshold component of the IS is the number of modify transactions after which RPS is to write the data buffers out to the data file on disk. A modify transaction is an insert, delete, or update operation.

Another way to think of this number is the number of modified records you're willing to lose if your system fails.

Reason for Supplying: (in bold type)

RPS uses the number you supply to determine when to flush the data from the buffers to the disk. The issue involved in choosing this number is one of performance versus data integrity.

If you specify a relatively low number, RPS will flush the buffers often, using time, but assuring greater integrity of the data. If you specify a higher number, RPS will flush the buffers to the disk less often, using less time, but risking the integrity of the data.

If you specify a write_threshold of 0 , RPS will determine when to flush the buffer.

The value for write_threahold must be greater than or equal to 0 .

Default Supplied by RPS: (in bold type) 1 transaction

@ extend_size

The extend_size component of the IS is the number of pages by which you wish to extend a file when there's no more space available in the file. This component applies to both data files and index files.

Reason for Supplying: (in bold type)

RPS uses the number you supply to determine how much space to add to your file when the file has used all its available space.

The value for extend_size must be greater than or equal to 1 .

Default Supplied by RPS: (in bold type) 4 pages

@ Summary of Buffer Management

The goal of the optimization process is to adjust the space/performance ratio to improve the performance of your application. By using the space allocation components of the IS, you make sure that RPS has enough space to hold the structures it needs, without using any unnecessary space. In conjunction with this, you use the buffer management components to manage the remaining available memory space for data records and key values.

@@@ RPS_INITIALIZE Call

Purpose: (in bold type)

RPS_INITIALIZE computes and allocates space for RPS internal use and for buffers to be used for paging. RPS uses the values of the IS components to compute the space needed.

Call Format: (in bold type)

RPS_INITIALIZE (pointer to IS)

Possible Status Returns:

0	Successful execution
-1101	Memory not available
-1201	Area already initialized
-1202	Invalid key size
-1203	Invalid record size
-1204	Invalid number of files
-1205	Invalid number of scans
-1206	Invalid number of indexes
-1207	Invalid number of segments
-1208	Invalid number of buffers
-1209	Invalid release priority
-1210	Invalid write threshold
-1211	Invalid file extend size
n	SOS error. Refer to Appendix B for an explanation of the status code.

Consequences of RPS_INITIALIZE: (in bold type)

RPS_INITIALIZE takes the numbers you specified in the IS, computes internal space requirements, and allocates the resulting space. This space is used for RPS internal structures and for buffers for paging RPS files.

If RPS tries to allocate more memory than is available in your Apple III, a fatal error occurs and no space is allocated. When you issue the next RPS call, RPS will allocate the amount of space it would have allocated by default had you never specified the RPS_INITIALIZE call. If there's not enough memory available for the default allocation, that call will also fail.

Additional Notes: (in bold type)

If you use RPS_INITIALIZE to allocate storage, that call must be the first RPS call in your program.

WARNING: Never use the Pascal MARK and RELEASE procedures and the RPS calls in this order:

```
MARK
first RPS call
.
.
.
RELEASE
.
.
.
last RPS call
```

If you do, Pascal will release the space taken by RPS for its internal structures.

Appendix E contains an example of the RPS_INITIALIZE call and a sample IS in the set_up_storage procedure of the complex program.

@@@ Summary Information

The rest of this chapter contains summarizes information about RPS optimization.

@@ What You Need to Know About Your Application Before You Specify the IS

Component	Facts Needed to Determine It
num_files	The number of files being scanned simultaneously at every instant in your application
num_scans	The number of scans open simultaneously at every instant in your application
num_segs	The different files being scanned simultaneously at every instant in your aapplication
	The number of key segments in every key definition for every file being scanned simultaneously
num_indexes	The different files being scanned simultaneously at every instant in your application, and, for each file, the type of processing requested, either modify or read-only
	The number of indexes associated with each different file
key_size	The key definitions that are being used for all the scans in your application
record_size	The sizes of all the records to be updated
num_512_buffs	The page sizes for the data records and indexes
num_1024_buffs	The page sizes for the data records and indexes
num_1536_buffs	The page sizes for the data records and indexes
num_2048_buffs	The page sizes for the data records and indexes

rel_priority	The number of levels RPS used to build all the indexes used for scans in your entire application
	The kind of processing, either random or sequential, done most often in the entire application
write_threshold	The number of "written" records (those that have been inserted, updated, or deleted) you are willing to lose if your system crashes
extend_size	The number of pages you want to extend your file by when the space currently allocated is full

Table 3-2. Summary of Facts Needed To Set Up the IS

@@ Summary of Default IS Values

Component	Value	Value Range
key_size	20	1 <= key_size <= 2048
record_size	100	1 <= record_size <= 2048
num_files	1	1 <= num_files
num_scans	2	1 <= num_scans
num_indexes	4	1 <= num_indexes <= num_files * 8
num_segs	16	1 <= num_segs <= num_files * 64
num_512_buffs	8	0 <= num_512_buffs
num_1024_buffs	0	0 <= num_1024_buffs
num_1536_buffs	0	0 <= num_1536_buffs
num_2048_buffs	0	0 <= num_2048_buffs
rel_priority	0,10,10,...,10	-1 <= rel_priority <= 100
write_threshold	1	0 <= write_threshold
extend_size	4	1 <= extend_size

@@ Summary of Equations for Space Allocation

Amount of Space for RPS File Descriptors: (in bold type)

$$S_{file} = n_{file} * (196 + 2 * 1024)$$

where S_{file} is the total number of bytes to be allocated

n_{file} is the number of files you specify

Amount of Space for RPS Scan Descriptors: (in bold type)

$$S_{scan} = n_{scan} (n_{ksize} + 22)$$

where S_{scan} is the total number of bytes to be allocated

n_{scan} is the number of scans you specify

n_{ksize} is the number you specify for the key_size component

Amount of Space for RPS Index Descriptors : (in bold type)

$$S_{index} = n_{index} * 44$$

where S_{index} is the total number of bytes to be allocated

n_{index} is the number of indexes you specify

Amount of Space for RPS Internal Segment Description Structures: (in bold type)

$$S_{seg} = n_{seg} * 16$$

where S_{seg} is the total number of bytes to be allocated

n_{seg} is the number of segments you specify

Amount of Space for RPS Buffers: (in bold type)

$$S_{buff} = n_{.5K}(526) + n_{1.0K}(1038) + n_{1.5K}(1550) + n_{2.0K}(2062)$$

where S_{buff} is the total number of bytes to be allocated

$n_{.5K}$ is the number of 512-byte buffers you specify

$n_{1.0K}$ is the number of 1024-byte buffers you specify

$n_{1.5K}$ is the number of 1536-byte buffers you specify

$n_{2.0K}$ is the number of 2048-byte buffers you specify

Total Amount of Space Allocated by RPS: (in bold type)

$$S_{total} = 4n_{ksize} + n_{rsize} + S_{file} + S_{scan} + S_{index} + S_{seg} + S_{buff}$$

@@@@ Appendix A

@@@ Glossary

Access Mode - The method of accessing records in an RPS file that determines the order in which the records are accessed. RPS provides six access modes: Next, Prior, First, Last, Current, and Direct.

Access Structure - A data structure in a program that is used to exchange information with RPS about an operation. You allocate and maintain the access structures necessary for the operations you want to perform. RPS provides eight access structures: the FAS, the KD, the SD, the FIS, the KIS, the SIS, the SAS, and the IS.

CROS - See Current Record of Scan.

Current Record of Scan (CROS) - An internal record pointer that points to a specific record, between two records, between the start-of-file and the first record, or between the end-of-file and the last record. RPS uses the CROS to maintain a current position within a file.

Damage Report Structure (DRS) - The data structure that you use to determine which areas of a file are damaged.

DRS - See Damage Report Structure.

Duplicate Key Value - A key value that is generated when two records have the same information in the portions of the record being used to form the key values.

FAS - See File Access Structure.

Field - Contiguous bytes of information grouped together into a meaningful unit.

File - A collection of related records.

File Access Structure (FAS) - The data structure that contains information about a file to be created or a file that already exists.

File Information Structure (FIS) - The data structure that you use to obtain information from RPS about a specific RPS file.

FIS - See File Information Structure.

Index - The set of all key values associated with a given key definition.

Initialization - The action of setting aside memory for internal use, which RPS performs when you issue the first RPS call in a program. RPS uses the Initialization Structure, if specified, to determine how much space it needs to interact with the program.

Initialization Structure (IS) - The data structure that RPS uses during the initialization process to determine how much space to allocate. RPS computes the space needed based on the values specified in the IS.

IS - See Initialization Structure.

KD - See Key Descriptor.

Key Area - The memory area where RPS is to find or place a key value.

Key Definition - The set of rules that tell RPS how to extract information from records to build key values and how to order the key values in the associated index.

Key Descriptor (KD) - The data structure that contains information about a key definition.

Key Identifier - A unique integer that RPS assigns to each key definition for a file when the file is created. You subsequently use this number to identify which key definition, and associated index, is to be used for a given scan. The Key Identifier also indicates the key definition of interest for the RPS_KEYINFO call.

Key Information Structure (KIS) - The data structure that you use to obtain information from RPS about a specific key definition.

Key Segment - The area within a record containing the data to be used in a key value.

Key Value - The information contained in a record that identifies that record within an RPS file.

KIS - See Key Information Structure.

Null Key Value - A key value generated when RPS extracts the data from all the key segments of a particular record, as specified in the key definition; collects the values together to form the record's key value; and ends up with a null value. A record with a null key value cannot be accessed using the index where the null value occurred because there is no reference to that record in the index.

Page - A unit of transfer of key values or of data records between a disk and memory.

Page Aging - The technique that RPS uses to ensure that the most frequently used pages from index files and data files stay in memory. RPS determines a page's age using the count registered on the internal RPS clock and the release priority assigned to the level of the page. The younger the page, the more likely it is to stay in memory.

RBS - See Rebuild Strcuture.

Rebuild Structure (RBS) - The data structure that provides RPS with information needed to reconstruct a damaged file.

Record - Fields of logically related information that have been grouped together.

Record Area - The area in memory where RPS is to find or place the data record.

Record Processing Services (RPS) - The file management system for the Apple III that provides a means of storing, accessing, and updating information in a file.

Release Priority - An array of values, specified in the IS, for data and for each level of an index. The purpose of using release priorities is to keep the most frequently used pages in memory. Release priorities make the pages on a given level appear older or younger to RPS than they really are.

RPS - See Record Processing Services.

SAS - See Scan Access Structure.

Scan - The process of moving from one record to another in an RPS file, in an order determined by program calls and associated access modes, for the purpose of selecting and manipulating a subset of records. A scan is associated with one and only one index.

Scan Access Structure (SAS) - The data structure that contains information about a record operation.

Scan Identifier - A unique integer that RPS assigns to a scan when the scan is opened. The Scan Identifier distinguishes a given scan from any other scan open on a file at a given time.

SD - See Segment Descriptor.

Segment Descriptor (SD) - The data structure that contains information about one key segment of a key definition.

Segment Identifier - A unique integer that RPS assigns to each key segment of a key definition when the associated file is created. The Segment Identifier indicates the key segment of interest for the RPS_SEGINFO call.

Segment Information Structure (SIS) - The data structure that you use to obtain information from RPS about a specific key segment.

SIS - See Segment Information Structure.

Sort Order - The ordering of records in a file, based on the ordering of the key values in an index associated with the file.

@@@@ Appendix B

@@@ Error Messages

This appendix contains all status returns that RPS issues, as well as the SOS errors that might occur while you're using RPS.

@@@ RPS Status Returns

For each RPS status return, we've listed the status code, the meaning of the code, the call or calls that issue the return, and, if the return is an error, some possible corrective action.

Status codes with negative values indicate that some kind of error has occurred. Status codes with positive values are informative only.

RPS returns status codes to your program via the status component of the access structures.

Status Code	Meaning	Suggested Corrective Action	Issued By
General Status Conditions			
1003	CROS is positioned at start-of-file.	None.	RPS_FETCH RPS_POSITION
1002	CROS is positioned at end-of-file.	None.	RPS_FETCH RPS_POSITION

1001	Record not found. Either the key value pointed to by the SAS did not exist in the index when you used Direct mode, or the CROS was not pointing to a record when you used Current mode.	None.	RPS_FETCH RPS_POSITION
0	Successful execution.	None.	All calls

Errors You Should Never Get

-1001 to -1010	Internal RPS errors.	If you get one of these error codes, you should note it and report it to Apple Computer, Inc.	

Space Related Errors

-1101	Memory not available. There is not enough stack-heap space for RPS to complete the current operation.	Use the IS and the RPS_INITIALIZE call to allocate more space.	All calls
-1102	No buffer available. RPS does not have enough buffers of the size needed.	Use the IS and the RPS_INITIALIZE call to allocate the right number of buffers.	All calls except RPS_CLOSESCAN
-1103	Volume full. There is no more room on the volume to handle the operation you're trying to perform.	Make more room on your volume.	RPS_INSERT RPS_UPDATE

Initialization Errors

-1201	RPS already initialized. The RPS_INITIALIZE call, if used, must be the first RPS call in the program.	Reorder the RPS calls so that the RPS_INITIALIZE call is the first RPS call in the program.	RPS_INITIALIZE
-1202	Invalid key size in the IS. The valid range of values is from 1 to 2048 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE

-1203	Invalid record size in the IS. The valid range of values is from 1 to 2048 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1204	Invalid number of files in the IS. The value must be greater than or equal to 1 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1205	Invalid number of scans in the IS. The value must be greater than or equal to 1 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1206	Invalid number of indexes in the IS. The valid range of values is from 1 to (num_files * 8).	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1207	Invalid number of segments in the IS. The valid range of values is from 1 to (num_files * 64).	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1208	Invalid number of buffers in the IS. The value must be greater than or equal to 0 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1209	Invalid release priority in the IS. The valid range of values is from -1 to 100 .	Check to make sure you typed the numbers correctly.	RPS_INITIALIZE
-1210	Invalid write threshold in the IS. The value must be greater than or equal to 0 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE
-1211	Invalid file extend size in the IS. The value must be greater than or equal to 1 .	Check to make sure you typed the number correctly.	RPS_INITIALIZE

File Creation Errors

-1301	Record size less than limit, which is 4 bytes.	Increase the record size.	RPS_CREATE RPS_INSERT RPS_UPDATE

-1302	Record size greater than limit, which is 2038 bytes for fixed length records and 2035 bytes for variable length records.	Decrease the record size.	RPS_CREATE RPS_INSERT RPS_UPDATE
-1303	Record/page size conflict The record size cannot exceed the data page size minus 10 for fixed length records. For variable length records, the record size cannot exceed the max rec size as computed from the formula given in the description of record_size in Chapter 2.	Change the record size or the data page size.	RPS_CREATE
-1304	Data page size less than limit. The page size must be at least 14 bytes for fixed length records and 17 bytes for variable length records.	Increase the data page size.	RPS_CREATE
-1305	Data page size greater than limit, which is 2048 bytes.	Decrease the data page size.	RPS_CREATE
-1306	Index page size less than limit, which is 34 bytes.	Increase the index page size.	RPS_CREATE
-1307	Index page size greater than limit, which is 2048 bytes.	Decrease the index page size.	RPS_CREATE
-1308	No pointer to KD in FAS. The FAS must have a pointer to at least one KD.	Add a pointer to a KD to the FAS.	RPS_CREATE
-1309	No pointer to SD in KD. A KD must have a pointer to at least one SD.	Add a pointer to an SD to the KD.	RPS_CREATE
-1310	Null use invalid. A KD specifies nulls allowed, but one of the key segments is of a type that cannot be null.	Change the segment or change the nulls allowed setting to nulls not allowed.	RPS_CREATE

-1311	Segment position invalid. The segment as described is not contained entirely in the record length specified in the FAS. Either the record size is wrong or the SD is wrong.	Change the record size or correct the SD.	RPS_CREATE
-1312	Segment size invalid.	Change the segment size.	RPS_CREATE
-1313	Number of key definitions greater than limit, which is 8 per file.	Restructure the file so that it contains no more than 8 key definitions.	RPS_CREATE
-1314	Number of key segments greater than limit, which is 8 per key definition.	Restructure the key definition so that it contains no more than 8 segments.	RPS_CREATE
-1315	Key/page size conflict. At least 4 key values plus 30 bytes must fit on an index page.	Refer to the formula for determining the number of key values per index page (found in Chapter 2). Change the key size or the index page size to accommodate this number.	RPS_CREATE

File Access Errors

-1401	File not RPS file. RPS found files with the appropriate data and index suffixes, but internal header information in these files was incorrect. Either the files have been corrupted or they were not RPS files.	Specify a correct RPS file name, or, if the files are corrupt, use the RPS_REBUILD procedure described in Appendix C to salvage as much data as possible.	RPS_FILEINFO RPS_KEYINFO RPS_SEGINFO RPS_DESTROY RPS_ERASE PPS_OPENSCAN
-1402	Pathname greater than limit. The maximum number of characters is 128 .	Change the path name so that it has no more than 128 characters.	RPS_FILEINFO RPS_KEYINFO RPS_SEGINFO RPS_DESTROY RPS_ERASE RPS_OPENSCAN

-1403	No data file. RPS found the index file, but not the data file.	Use your backup copy of the RPS file.	RPS_FILEINFO RPS_KEYINFO RPS_SEGINFO RPS_DESTROY RPS_ERASE RPS_OPENSCAN
-1404	No index file. RPS found the data file, but not the index file.	Use the RPS_REBUILD procedure described in Appendix C to create a new index file.	RPS_FILEINFO RPS_KEYINFO RPS_SEGINFO RPS_DESTROY RPS_ERASE RPS_OPENSCAN
-1405	Corrupt data. The data file has been corrupted.	Use the RPS_REBUILD procedure described in Appendix C to salvage as much data as possible.	All calls except RPS_CREATE RPS_DESTROY RPS_ERASE
-1406	Corrupt index. The index file has been corrupted.	Use the RPS_REBUILD procedure described in Appendix C to create a new index.	All calls except RPS_CREATE RPS_DESTROY RPS_ERASE
-1407	Key Identifier invalid. The key_id specified in the access structure does not exist for this file.	Use the RPS_FILEINFO call to determine how many indexes the file has.	RPS_KEYINFO RPS_SEGINFO RPS_OPENSCAN
-1408	Segment Identifier invalid. The seg_id specified in the access structure does not exist for this key definition.	Use the RPS_KEYINFO call to determine how many segments the key definition has.	RPS_SEGINFO
-1409	Scan Identifier invalid. The scan_id specified in the SAS must be -1 when a scan is opened, and it must be equal to the number set by RPS for all other record level calls.	Correct the scan_id value.	RPS_OPENSCAN RPS_CLOSESCAN RPS_FETCH RPS_POSITION RPS_INSERT RPS_UPDATE RPS_DELETE
-1410	Number of files greater than number allowed for during initialization. You tried to access more files than RPS expected, based on the value of num_files in the IS or the RPS default.	Change the value specified for num_files in the IS.	RPS_FILEINFO RPS_KEYINFO RPS_SEGINFO RPS_CREATE RPS_DESTROY RPS_ERASE RPS_OPENSCAN

-1411	Number of indexes greater than number allowed for during initialization. You tried to use more indexes than RPS expected, based on the value of num_indexes in the IS or the RPS default.	Change the value specified for num_indexes in the IS.	RPS_OPENSCAN
-1412	Number of scans greater than number allowed for during initialization. You tried to open more scans than RPS expected, based on the value of num_scans in the IS or the RPS default.	Change the value specified for num_scans in the IS.	RPS_OPENSCAN

Record Processing Errors

-1501	CROS position invalid. You tried to update or delete a record when the CROS was not pointing directly to a record.	Position the CROS correctly.	RPS_UPDATE RPS_DELETE
-1502	Record in use by another scan. The contents of the record might be changed.	None.	RPS_FETCH RPS_POSITION
-1503	Record locked by another scan. You cannot access the record until the other scan unlocks it.	Wait until the other scan unlocks the record.	RPS_FETCH RPS_POSITION
-1504	Processing attempted not allowed. You tried to insert, update, or delete a record with a scan open for read-only processing.	Change the process_ required component of the SAS associated with the scan.	RPS_INSERT RPS_UPDATE RPS_DELETE
-1505	No path to record. All key definitions allow null key values and all key values are nulls. RPS does not insert the record.	None.	RPS_INSERT RPS_UPDATE

-1506	File in use. You tried to open a scan on a file for exclusive use, but the file was already open.	Reexamine the coordination of the scans.	RPS_OPENSCAN
-1507	File locked. You tried to open a scan on a file that was already open for exclusive use by another scan.	Reexamine the coordination of the scans.	RPS_OPENSCAN
-1550 to -1557	No duplicates allowed. You tried to put a record into the file that has a key value identical to that of a record already in the file. The key definition for this key value does not allow this for the specified index.	Look at the rightmost digit; it gives the Key Identifier of the index that caused the error. Supply a unique key value for the record for this key definition.	RPS_INSERT RPS_UPDATE
-1560 to -1567	No nulls allowed. You tried to put a record into the file that has a key value that is null. The key definition for this key value does not allow this for the specified index.	Look at the rightmost digit; it gives the Key Identifier of the index that caused the error. Supply a nonnull key value for the record for this key definition.	RPS_INSERT RPS_UPDATE
-1570 to -1577	Index too large. The data file has so many records that the associated index exceeds the maximum allowed space for key values (2 megabytes).	Look at the rightmost digit; it gives the Key Identifier of the index that caused the error. Divide the information into more than one RPS file or reduce the key value size for the index.	RPS_INSERT RPS_UPDATE
-1506	Index level too deep. The data file has so many records that the associated index cannot hold the key values for all the records.	Look at the rightmost digit; it gives the Key Identifier of the index that caused the error. Divide the information into more than one RPS file or reduce the key value size for the index.	RPS_INSERT RPS_UPDATE

@@@ SOS Error Messages

In addition to the RPS status codes, RPS returns SOS errors. The SOS errors that RPS can return are listed below. Note that RPS places a minus sign in front of each SOS code.

For full explanations of these errors, consult the Apple III SOS Reference Manual.

Status Code		Meaning
General System Errors		
-01	BADSCNUM	Invalid SOS call number.
-02	BADCZPAGE	Invalid caller zero page.
-03	BADXBYTE	Invalid indirect pointer X-byte.
-04	BADSCPCNT	Invalid SOS call parameter count.
-05	BADSCBNDS	SOS call pointer out of bounds.
Device System Errors		
-16	DNFERR	Device not found.
-17	BADDNUM	Invalid device number.
-32	BADREQCODE	Invalid request code.
-33	BADCTL	Invalid status or control code.
-34	BADCTLPARM	Invalid control parameter list.
-35	NOTOPEN	Device not open.
-37	NORESRC	Resources not available.
-38	BADOP	Call not supported on device.
-39	IOERROR	I/O error.
-43	NOWRITE	Device write-protected.
-46	DISKSW	Disk switched.
-48 through -63		Device-specific error.
File System Errors		
-64	BADPATH	Invalid pathname syntax.
-65	CFCBFULL	Character File Control Block full.
-66	FCBFULL	Block File or Volume Control Block full.
-67	BADREFNUM	Invalid reference number.
-68	PNFERR	Path not found.
-69	VNFERR	Volume not found.
-70	FNFERR	File not found.
-71	DUPERR	Duplicate file name.
-72	OVRERR	Overrun on volume.
-73	DIRFULL	Directory full.
-74	CPTFRR	Incompatible file format.
-75	TYPERR	Unsupported storage type.
-76	EOFERR	End-of-file would be exceeded.
-77	POSNERR	Position out of range.
-78	ACCSERR	Access not allowed.

-79	BTSERR	Buffer too small.
-80	FILBUSY	File busy.
-81	DIRERR	Directory error.
-82	NOTSOS	Not a SOS volume.
-83	BADLSTCNT	Length parameter invalid.
-84	OUTOFMEM	Out of memory.
-85	BUFTBLFULL	Buffer table full.
-86	BADSYSBUFF	Invalid system buffer parameter.
-87	DUPVOL	Duplicate volume.
-88	NOTBLKDEV	Not a block device.
-89	LVLERR	Invalid level.
-90	BITMAPADDR	Invalid bit map address.

Miscellaneous Errors

-112	BADJMODE	Invalid joy_mode parameter.

Memory System Errors

-224	BADBKPG	Invalid segment address.
-225	SEGRQDN	Segment request denied.
-226	SEGTBLFULL	Segment table full.
-227	BADSEGNUM	Invalid segment number.
-228	SEGNOTFND	Segment not found.
-229	BADSRCHMODE	Invalid search_mode parameter.
-230	BADCHGMODE	Invalid change_mode parameter.
-231	BADPGCOUNT	Invalid pages parameter.

@@@@ Appendix C

@@@ Repairing a Damaged RPS File

This appendix describes what to do when you discover that an RPS file has been damaged. With special access structures and program calls described in this appendix, you can return a damaged RPS file to its previous undamaged state.

This appendix has two parts.

> The first part contains general information about what a damaged file is and how you can fix a damaged file.
>
> The second part describes the program calls and access structures to find out what's wrong with a file and to subsequently rebuild the file.

@@@ General Information

This section describes what a damaged file is and the steps to take to repair a damaged file.

@@ What Is a Damaged File?

A damaged file is one whose structure or content, or both, have been altered in some unintended way. The data in a damaged file can no longer be considered reliable. When RPS suspects that a file has been damaged, it marks the file as damaged; you must repair the file before using it again in normal processing.

RPS tells you that a file has been damaged when you issue a call using that file. The following status returns indicate file damage:

-1401	File not RPS file
-1403	No data file
-1404	No index file
-1405	Corrupt data
-1406	Corrupt index

If RPS issues any of these status returns, except status return -1403, you must repair your file before continuing normal RPS processing. If RPS issues status return -1403, which means that no data file exists, the RPS file cannot be reconstructed unless you have a backup copy of the file.

Damage Areas. Damage to an RPS file can occur in one or more of the following areas:

the internal header information of the data file
the internal header information of the index file
the copy of the FAS in the file
one or more of the KD copies
one or more of the SD copies
the data portion of the data file
one or more of the indexes of the index file

The next section describes how to find out which area ia damaged and how to fix it.

@@ How To Repair a Damaged File

The steps for repairing a damaged file are as follows:

1. Get a damage report on the file with the RPS_DAMAGE_REPORT call.

   The RPS_DAMAGE_REPORT call uses the Damage Report Structure (DRS) to return information about the areas of a file that could possibly be damaged. For each of these areas, RPS returns a true or false damage status.

2. Check the damage report to see if copies of the FAS or any of the KDs or SDs are damaged.

   If none of these structures is damaged, proceed to step 4.

   If any of these structures is damaged, find out which part of the sturcture is damaged with one or more of the following calls:

   RPS_DMG_FILEINFO
   RPS_DMG_KEYINFO
   RPS_DMG_SEGINFO

   These calls return the same information as the RPS_FILEINFO, RPS_KEYINFO, and RPS_SEGINFO calls, but they have the additional feature of working on a damaged file. The other information calls don't work on a damaged file.

3. Fix the damaged part of the structure.

   You must recreate the structure of the FAS and the linked lists of the KDs and SDs. Then supply correct values for the damaged structures. It is not necessary to supply any value other than the pointer value for undamaged structures. This is because they serve only as place holders to get to the damaged structures.

   Note that when RPS rebuilds a file, RPS works only with the damaged structures and does nothing with the undamaged structures. Consequently, any changes you make to the undamaged structures will be ignored.

4. Rebuild the file with the RPS_REBUILD call.

   The RPS_REBUILD call uses the Rebuild Structure (RBS) to get the name of the damaged file, the name of the new file to be built, the name of a discard file, and the location of the corrected access structures. RPS uses the discard file to hold any data that's of questionable validity.

@@@ Repairing the File

RPS provides five program calls to use in the file repair process.

Program Call	Function
RPS_DAMAGE_REPORT	returns information about which areas of a file have been damaged
RPS_DMG_FILEINFO	returns information about a specific file
RPS_DMG_KEYINFO	returns information about a specific key definition
RPS_DMG_SEGINFO	returns information about a specific key segment
RPS_REBUILD	reconstructs the damaged areas of an RPS file

@@ Building the File Repair Access Structures

This section describes the access structures for repairing a damaged RPS file. The structures and the calls that use them are listed below.

Structure	Used By
Damage Report Structure (DRS)	RPS_DAMAGE_REPORT
Rebuild Structure (RBS)	RPS_REBUILD

In addition to these structures, RPS uses the FIS, the KIS, and the SIS in the RPS_DMG_FILEINFO, RPS_DMG_KEYINFO, and RPS_DMG_SEGINFO calls. The FIS, KIS, and SIS are described in Chapter 2.

@ Damage Report Structure

The Damage Report Structure (DRS) provides you with detailed information on a damaged file. The DRS is used by RPS_DAMAGE_REPORT. Table C-1 shows the layout of the DRS.

Component	Data Type	Supplied By	Comments
damaged_ filename_ref	^rps_data	user to RPS	A pointer to a file name area.
data_file_ missing	Boolean	RPS to user	The setting that indicates whether or not the data file is missing.
data_file_ not_rps	Boolean	RPS to user	The setting that indicates whether or not the internal header information of the data file has been damaged.
index_file_ missing	Boolean	RPS to user	The setting that indicates whether or not the index file is missing.
index_file_ not_rps	Boolean	RPS to user	The setting that indicates whether or not the internal header information of the index file has been damaged.
FAS_damaged	Boolean	RPS to user	The setting that indicates whether or not the copy of the FAS has been damaged.
KD_damaged	packed array [0..7] of Boolean	RPS to user	The settings that indicate whether or not one or more of the KD copies have been damaged.
SD_damaged	packed array [0..7,0..7] of Boolean	RPS to user	The settings that indicate whether or not one or more of the SD copies have been damaged.
data_corrupted	Boolean	RPS to user	The setting that indicates whether or not data in the file has been corrupted.

index_corrupted	packed array [0..7] of Boolean	RPS to user	The settings that indicate whether or not one or more of the indexes have been corrupted.
status	integer	RPS to user	The status of the operation upon completion.

Table C-1. DRS Layout

Notes on DRS Components

SD_damaged

This component is a two-dimensional array whose first index represents the Key Identifiers and whose second index represents the Segment Identifiers. For example, the statement

```
SD_damaged [2,6] = true
```

means that the seventh segment (seg_id = 6) of the third key definition (key_id = 2) is damaged.

@ Rebuild Structure

The Rebuild Structure (RBS) supplies RPS with the information needed to reconstruct the RPS file. You use the RBS in conjunction with the RPS_REBUILD call. Table C-2 shows the layout of the RBS.

Component	Data Type	Supplied By	Comments
damaged_ filename_ref	^rps_data	user to RPS	A pointer to a file name area for the damaged file.
new_ filename_ref	^rps_data	user to RPS	A pointer to a file name area for the new file to be built.
discard_ filename_ref	^rps_data	user to RPS	A pointer to a file name area for the discard file.
corrected_ FAS_ref	^FAS	user to RPS	A pointer to an area with the corrected access structures for the file to be built.
discard_limit	array [0..1] of integer	user to RPS	The maximum number of records RPS is to discard before abnormally terminating the RPS_REBUILD procedure.
status	integer	RPS to user	The status of the operation upon completion.

Table C-2. RBS Layout

Notes on RBS Components

damaged filename ref

The file name pointed to is the name of the file that has been damaged.

new filename ref

The file name pointed to is the name of the file that RPS will build from the damaged file.

discard filename ref

The file name pointed to is the name of an ASCII file into which RPS will dump records from the damaged file that have questionable validity.

corrected FAS ref

The FAS pointed to is the one RPS will use to rebuild the file. The FAS points to a list of KDs, and each KD points to a list of SDs. These are the same structures RPS used to create the file.

You must recreate the structure of the FAS and the linked lists of the KDs and SDs. Then supply correct values for the damaged structures. It is not necessary to supply any value other than the pointer value for undamaged structures. This is because they serve only as place holders to get to the damaged structures.

In addition, because RPS assumes that the undamaged structures contain the same information as before, any changes you make to them will be ignored.

If none of the structures for the file have been damaged, you need only recreate the structure of the FAS and the linked lists of KDs and SDs.

@@ File Repair Calls

The following sections describe the five procedures that you use to repair a damaged file. The procedures are

RPS_DAMAGE_REPORT
RPS_DMG_FILEINFO
RPS_DMG_KEYINFO
RPS_DMG_SEGINFO
RPS_REBUILD

@ RPS_DAMAGE_REPORT

Purpose: (in bold type)

The RPS_DAMAGE_REPORT call gives you information about whether or not any areas of an RPS file have been damaged. The call uses the DRS to transfer the information.

Call Format: (in bold type)

RPS_DAMAGE_REPORT (pointer to DRS)

Components of DRS Required for Call: damaged_filename_ref

Possible Status Returns:

0	Successful execution
-1801	Damaged RPS file
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DAMAGE_REPORT: (in bold type)

When you call for a damage report, RPS checks for damage to all areas of the specified file. For any area that has been marked damaged, RPS returns a value of true in the appropriate DRS component.

@ RPS_DMG_FILEINFO

Purpose: (in bold type)

The RPS_DMG_FILEINFO call transfers information to you about an RPS file. The call returns the same information as the RPS_FILEINFO call. The only difference between the two calls is that RPS_DMG_FILEINFO will work on a file that is marked damaged, and RPS_FILEINFO will not.

Call Format: (in bold type)

RPS_DMG_FILEINFO (pointer to FIS)

Components of FIS Required for Call: filename_ref

Possible Status Returns:

0	Successful execution
-1403	Data file missing
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DMG_FILEINFO: (in bold type)

When you issue an RPS_DMG_FILEINFO call, RPS fills the FIS with the same type of information returned with the RPS_FILEINFO call, even if any part of the specified file has been damaged. Check the components of the FIS to try to determine which components of the FAS are damaged.

@ RPS_DMG_KEYINFO

Purpose: (in bold type)

The RPS_DMG_KEYINFO call transfers information to you about a key definition for an RPS file. The call returns the same information as the RPS_KEYINFO call. The only difference between the two calls is that RPS_DMG_KEYINFO will work on a file that is marked damaged, and RPS_KEYINFO will not.

Call Format: (in bold type)

RPS_DMG_KEYINFO (pointer to KIS)

Components of KIS Required for Call:	filename_ref key_id

Possible Status Returns:

0	Successful execution
-1404	Index file missing
-1407	Key Identifier invalid
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DMG_KEYINFO: (in bold type)

When you issue an RPS_DMG_KEYINFO call, RPS fills the KIS with the same type of information returned with the RPS_KEYINFO call, even if any part of the specified file has been damaged. Check the components of the KIS to try to determine which components of the KD are damaged.

@ RPS_DMG_SEGINFO

Purpose: (in bold type)

The RPS_DMG_SEGINFO call transfers information to you about a key segment for an RPS file. The call returns the same information as the RPS_SEGINFO call. The only difference between the two calls is that RPS_DMG_SEGINFO will work on a file that is marked damaged, and RPS_SEGINFO will not.

Call Format: (in bold type)

RPS_DMG_SEGINFO (pointer to SIS)

Components of SIS Required for Call:	filename_ref key_id seg_id

Possible Status Returns:

0	Successful execution
-1404	Index file missing
-1407	Key Identifier invalid
-1408	Segment Identifier invalid
n	SOS error. Refer to Appendix B for an explanation of this status code.

Consequences of RPS_DMG_SEGINFO: (in bold type)

When you issue an RPS_DMG_SEGINFO call, RPS fills the SIS with the same type of information returned with the RPS_SEGINFO call, even if any part of the specified file has been damaged. Check the components of the SIS to try to determine which components of the SD are damaged.

@ RPS_REBUILD

Purpose: (in bold type)

The RPS_REBUILD call reconstructs a damaged RPS file and returns it to an undamaged state. The new file may possibly have some records omitted.

Call Format: (in bold type)

RPS_REBUILD (pointer to RBS)

Components of RBS Required for Call:

damaged_filename_ref
new_filename_ref
discard_filename_ref
corrected_FAS_ref

Possible Status Returns:

0	Rebuild successful with no discarded records
-1301 to -1315	All errors possible during file creation
-1403	Data file missing
-1802	Rebuild successful with discarded records
-1803	Discard limit reached
-1804	File too damaged to repair

Consequences of RPS_REBUILD: (in bold type)

When you issue an RPS_REBUILD call and any of the structures are damaged, RPS first looks through the corrected FAS and associated KDs and SDs for the information needed to repair those structures.

RPS then begins to repair the data file, if damaged. RPS reads through the data file and tries to determine if a record is valid. If the record appears to be valid, RPS places it in the new RPS data file. If the record appears to be invalid, RPS places it in the discard file. After reconstructing the data, or if the data area was undamaged, RPS rebuilds all damaged indexes.

WARNING: If the data area was undamaged, the new data file created by the RPS_REBUILD call is identical to the old file before it was damaged. If the data area was damaged, you cannot be certain that the new RPS file contains all valid records. Therefore, some other validity check is necessary.

If RPS does not discard any records, no discard file is created.

@@@@ Appendix D

@@@ Summary of Specifications and Restrictions

@@@ RPS Restrictions

Key Definitions (in bold type)

Number of Key Definitions: minimum of one, maximum of eight.

Number of Segments Per Key Definition: minimum of one, maximum of eight.

Maximum Key Size: 504 bytes.

Indexes (in bold type)

Maximum Space for Key Values: 2 megabytes per index.

Maximum Number of Index Levels: 10 levels.

Records (in bold type)

Maximum Record Size: 2K bytes for both fixed and variable length records.

Number of Records Per File: depends on record size and key size, constrained by limits on data and index space.

Files (in bold type)

Number of Files Open Simultaneously: see SOS Restrictions.

Number of Scans Open Simultaneously: constrained by memory; no architectural limit.

Maximum Space for Data: 16 megabytes.

Maximum Page Size: 2048 bytes.

Minimum Page Size: 14 bytes.

@@@ SOS Restrictions

Number of Files Open Simultaneously: 6 RPS files, which is equivalent to 12 SOS files.

Maximum File Size: 16 megabytes.

@@@ Appendix E

@@@@ Sample Pascal Applications

@@@ Simple Example

```
uses  rps;

type
    data = record
```

```
        part_num : integer;
        price : real;
        num_in_stock : integer;
        description : string[20];
    end;

var
    part_file : string[30];
    FAS_ref : ptr_FAS;
    SAS_ref : ptr_SAS;
    key : integer;
    rec : data;

procedure check_for_error;
    begin
        if SAS_ref^.status < 0 then begin
            RPS_CLOSESCAN(SAS_ref);
            exit(program);
            end;
    end;

begin
    part_file := '/PROFILE/PARTS';

  { Set up FAS }
    new(FAS_ref);
    with FAS_ref^ do begin
        filename_ref := RPS_PTR_LOAD(part_file);
        record_size := sizeof(data);
        record_type := fixed_length;
        data_page_size := 512;
        new(KD_ref);

  { Define key on field num_in_stock }
        with KD_ref^ do begin
            dupls_allowed := true;
            nulls_allowed := false;
            nulls_kept := false;
            index_page_size := 512;
            next_KD_ref := nil;
            new(SD_ref);
            with SD_ref^ do begin
                pos_in_record := sizeof(integer)
                                 + sizeof(real);
                seg_size := sizeof(integer);
                seg_type := seg_ints16;
                sort_direction := ascending;
                next_SD_ref := nil;
                end;
            end;
        end;

  { Create file }
    RPS_CREATE(FAS_ref);
```

```
  if FAS_ref^.status <> 0 then exit(program);

{  Open scan on file using num_in_stock key  }
  new(SAS_ref);
  with SAS_ref^ do begin
      scan_id := -1;
      filename_ref := RPS_PTR_LOAD(part_file);
      file_use := exclusive;
      process_required := modify;
      access_mode := next;
      record_locking := false;
      key_id := 0;
      record_area_ref := RPS_PTR_LOAD(rec)
      end;
  RPS_OPENSCAN(SAS_ref);
  if SAS_ref^.status <> 0 then exit(program);

{  Collect part information and add to file  }
  with rec do
      repeat
          write('part number ? ');
          readln(part_num);
          if 0 < part_num then begin
              write('price of part ? ');
              readln(price);
              write('number in stock ? ');
              readln(num_in_stock);
              write('description of part ? ');
              readln(description);
              RPS_INSERT(SAS_ref);
              check_for_error;
              end;
          until part_num <= 0;

{  Print list of parts by number in stock  }
  SAS_ref^.access_mode := first;
  RPS_FETCH(SAS_ref);
  check_for_error;
  SAS_ref^.access_mode := next;
  RPS_FETCH(SAS_ref);
  check_for_error;
  while SAS_ref^.status <> 1002 do begin
      with rec do
          writeln(part_num:5,' $',price:5:2,
                  num_in_stock:5,description);
      RPS_FETCH(SAS_ref);
      check_for_error;
      end;

{  Close scan on part file  }
  RPS_CLOSESCAN(SAS_ref);

end.
```

@@@ Complex Example

```
{-------------------------------------------------------------------------}
{                                                                         }
{    The following RPS example program demonstrates the use of each RPS   }
{    call.  This program creates a file in the subdirectory PAYROLL on    }
{    the disk volume PROFILE.  It then proceeds to populate the file      }
{    by soliciting employee data from the console.  After the file is     }
{    populated, a number of lists of information are printed to the       }
{    console.  Next, particular employees are deleted and the records     }
{    of the remaining employees are updated.  Finally, the employee file  }
{    is erased, repopulated, and then destroyed.                          }
{                                                                         }
{-------------------------------------------------------------------------}

program RPS_complex_example;

uses  rps;

const
        no_error = 0;
        eof_msg = 1002;

type
        byte = 0..255;

        date =
            packed record
                year : integer;
                month : byte;
                day : byte;
            end;

        name = string[12];

        number = packed array [1..9] of byte;

        data =
            record
                employee_id : integer;
                first_name : name;
                last_name : name;
                age : integer;
                sex : char;
                birthday : date;
                citizenship : char;
                salary : real;
                ss_number : number;
                manager_id : integer;
            end;
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section checks for a fatal RPS error.  If one has occurred, the }
{    program execution is terminated.                                    }
{                                                                        }
{------------------------------------------------------------------------}

procedure  check_for_error ( status : integer );

begin
        if status < no_error then begin
            writeln(' Error in RPS call.  Status Number = ',status:5);
            exit(program);
            end;

end;  { check_for_error }
```

```
{-------------------------------------------------------------------------}
{                                                                         }
{    This section allocates the memory space needed for RPS to execute.  }
{                                                                         }
{-------------------------------------------------------------------------}

procedure  set_up_storage;

var
        IS_ref : ptr_IS;
        i : integer;

begin
        new(IS_ref);
        with IS_ref^ do begin
            key_size := 4;
            record_size := 56;
            num_files := 1;
            num_scans := 2;
            num_indexes := 5;
            num_segs := 7;
            num_512_buffs := 8;
            num_1024_buffs := 4;
            num_1536_buffs := 0;
            num_2048__buffs := 0;
            for i := 0 to 10 do rel_priority[i] := 10;
            write_threshold := 10;
            extend_size := 6;
            end;
        RPS_INITIALIZE(IS_ref);
        check_for_error(IS_ref^.status);

end;  { set_up_storage }
```

This section creates an RPS file with the given layouts.

Record layout:

Field Name	Field Type	Field Size	Field Position
Employee_id	Integer	2	0
First_name	Name	14	2
Last_name	Name	14	16
Age	Integer	2	30
Sex	Character	2	32
Birthday	Date	4	34
Citizenship	Character	2	38
Salary	Real	4	40
SS_number	Number	10	44
Manager_id	Integer	2	54

Key layout:

Key	Seg	Seg Position	Seg Size	Seg Type	Duplicate	Null
0	0	0	2	Integer	False	False
1	0	44	3	Byte Array	True	False
	1	30	2	Integer		
	2	40	4	Real		
2	0	34	4	Byte Array	True	False
3	0	40	4	Real	True	False
4	0	38	2	Char Array	True	True

Key definitions and record fields are related as follows:

Key	Seg	Associated Field
0	0	Employee_id
1	0	SS_number (first three digits only)
	1	Age
	2	Salary
2	0	Birthday
3	0	Salary
4	0	Citizenship

```
procedure  make_employee_file;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        FAS_ref : ptr_FAS;
        current_KD_ref : ptr_KD;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up FAS.  }

        new(FAS_ref);
        with FAS_ref^ do begin
            filename_ref := RPS_PTR_LOAD(name_of_file);
            record_size := sizeof(data);
            record_type := fixed_length;
            data_page_size := 1024;
            new(KD_ref);
            current_KD_ref := KD_ref;
            end;

        {  Define first key.  Key Identifier is 0 .}

        with current_KD_ref^ do begin
            dupls_allowed := false;
            nulls_allowed := false;
            nulls_kept := false;
            index_page_size := 512;
            new(next_KD_ref);
            new(SD_ref);
            with SD_ref^ do begin
                pos_in_record := 0;
                seg_size := sizeof(integer);
                seg_type := seg_intu16;
                sort_direction := ascending;
                next_SD_ref := nil;
                end;
            end;
        current_KD_ref := current_KD_ref^.next_KD_ref;

        {  Define second key.  Key Identifier is 1 .}

        with current_KD_ref^ do begin
            dupls_allowed := true;
            nulls_allowed := false;
            nulls_kept := false;
            index_page_size := 1024;
            new(next_KD_ref);
            new(SD_ref);
            with SD_ref^ do begin
```

```
            pos_in_record := sizeof(data) -
                             sizeof(integer) -
                             sizeof(number);
            seg_size := 3;
            seg_type := seg_packbyte;
            sort_direction := ascending;
            new(next_SD_ref);
            with next_SD_ref^ do begin
                pos_in_record := sizeof(integer) + 2*sizeof(name);
                seg_size := sizeof(integer);
                seg_type := seg_intu16;
                sort_direction := descending;
                new(next_SD_ref);
                with next_SD_ref^ do begin
                    pos_in_record := sizeof(data) -
                                     sizeof(integer) -
                                     sizeof(number) -
                                     sizeof(real);
                    seg_size := sizeof(real);
                    seg_type := seg_real32;
                    sort_direction := ascending;
                    next_SD_ref := nil;
                    end;
                end;
            end;
        end;
    current_KD_ref := current_KD_ref^.next_KD_ref;

    { Define third key.  Key Identifier is 2 .}

    with current_KD_ref^ do begin
        dupls_allowed := true;
        nulls_allowed := false;
        nulls_kept := false;
        index_page_size := 512;
        new(next_KD_ref);
        new(SD_ref);
        with SD_ref^ do begin
            pos_in_record := 2*sizeof(integer) +
                             2*sizeof(name) +
                             sizeof(char);
            seg_size := sizeof(date);
            seg_type := seg_packbyte;
            sort_direction := ascending;
            next_SD_ref := nil;
            end;
        end;
    current_KD_ref := current_KD_ref^.next_KD_ref;

    { Define fourth key.  Key Identifier is 3 .}

    with current_KD_ref^ do begin
        dupls_allowed := true;
        nulls_allowed := false;
```

```
            nulls_kept := false;
            index_page_size := 512;
            new(next_KD_ref);
            new(SD_ref);
            with SD_ref^ do begin
                pos_in_record := sizeof(data) -
                                 sizeof(integer) -
                                 sizeof(number) -
                                 sizeof(real);
                seg_size := sizeof(real);
                seg_type := seg_real32;
                sort_direction := ascending;
                next_SD_ref := nil;
                end;
            end;
        current_KD_ref := current_KD_ref^.next_KD_ref;

        {  Define fifth key.  Key Identifier is 4 .}

        with current_KD_ref^ do begin
            dupls_allowed := true;
            nulls_allowed := true;
            nulls_kept := false;
            index_page_size := 512;
            next_KD_ref := nil;
            new(SD_ref);
            with SD_ref^ do begin
                pos_in_record := 2*sizeof(integer) +
                                 2*sizeof(name) +
                                 sizeof(char) +
                                 sizeof(date);
                seg_size := sizeof(char);
                seg_type := seg_packchar;
                sort_direction := descending;
                next_SD_ref := nil;
                end;
            end;

        {  Create file.  }

        RPS_CREATE(FAS_ref);
        check_for_error(FAS_ref^.status);

        release(heap_ptr);

end;  { make_file }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section lists all the file characteristics of the employee file }
{    just created.                                                       }
{                                                                        }
{------------------------------------------------------------------------}

procedure  list_file_characteristics;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        FIS_ref : ptr_FIS;
        KIS_ref : ptr_KIS;
        SIS_ref : ptr_SIS;
        key_num : integer;
        seg_num : integer;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up information structures.  }

        new(FIS_ref);
        FIS_ref^.filename_ref := RPS_PTR_LOAD(name_of_file);

        new(KIS_ref);
        KIS_ref^.filename_ref := RPS_PTR_LOAD(name_of_file);

        new(SIS_ref);
        SIS_ref^.filename_ref := RPS_PTR_LOAD(name_of_file);

        {  Get and print file information.  }

        with FIS_ref^ do begin
            RPS_FILEINFO(FIS_ref);
            check_for_error(status);
            writeln('File Information');
            writeln('  Record Size = ',record_size);
            writeln('  Record Type = ',ord(record_type));
            writeln('  Record Count = ',record_count[0]);
            writeln('  Data Page Size = ',data_page_size);
            writeln('  Number of Indexes = ',num_indexes);
            end;

        {  Get and print key and segment information.  }

        for key_num := 0 to FIS_ref^.num_indexes-1 do
            with KIS_ref^ do begin
                key_id := key_num;
                RPS_KEYINFO(KIS_ref);
                check_for_error(status);
```

```
                writeln('Key Information');
                writeln('  Key ID = ',key_id);
                writeln('  Duplicates Allowed = ',ord(dupls_allowed));
                writeln('  Nulls Allowed = ',ord(nulls_allowed));
                writeln('  Nulls Kept = ',ord(nulls_kept));
                writeln('  Index Page Size = ',index_page_size);
                writeln('  Number of Segments = ',num_segs);
                writeln('  Key Count = ',key_count[0]);
                writeln('  Index Depth = ',index_depth);
                for seg_num := 0 to num_segs-1 do
                    with SIS_ref^ do begin
                        key_id := key_num;
                        seg_id := seg_num;
                        RPS_SEGINFO(SIS_ref);
                        check_for_error(status);
                        writeln('  Key ID = ',key_id);
                        writeln('  Segment ID = ',seg_id);
                        writeln('  Position in Key = ',pos_in_key);
                        writeln('  Position in Record = ',pos_in_record);
                        writeln('  Segment Size = ',seg_size);
                        writeln('  Segment Type = ',ord(seg_type));
                        writeln('  Sort Direction = ',ord(sort_direction));
                        end;
            end;

        release(heap_ptr);

end;  { list_file_characteristics }
```

```
{-------------------------------------------------------------------------}
{                                                                         }
{    This section forms employee records by collecting all relevant       }
{    information from the console.  The new record is then inserted into  }
{    the employee file.  The input process is terminated when a negative  }
{    employee id is encountered.                                          }
{                                                                         }
{-------------------------------------------------------------------------}

procedure  add_employees;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        rec : data;
        good_rec : boolean;
        value : integer;
        i : integer;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Open scan to insert records.  }

        new(SAS_ref);
        with SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := modify;
            access_mode := next;
            record_locking := false;
            key_id := 0;
            record_area_ref := RPS_PTR_LOAD(rec)
            end;
         RPS_OPENSCAN(SAS_ref);
         check_for_error(SAS_ref^.status);

         {  Collect employee data from screen and insert into file.  }

         good_rec := true;
         repeat
             with rec do begin
                 writeln;
                 write('  employee_id ? ');
                 readln(employee_id);
                 if employee_id < 0
                     then good_rec := false
                     else begin
                         write('  first_name ? ');  readln(first_name);
                         write('  last_name ? ');   readln(last_name);
```

```
                        write('  age ? ');          readln(age);
                        write('  sex ? ');          readln(sex);
                        writeln('  birthday');
                        write('    day ? ');        readln(value);
                        birthday.day := value;
                        write('    month ? ');      readln(value);
                        birthday.month := value;
                        write('    year ? ');       readln(birthday.year);
                        write('  citizenship ? '); readln(citizenship);
                        write('  salary ? ');       readln(salary);
                        write('  manager_id ? ');   readln(manager_id);
                        write('  ss_number ? ');
                        for i := 1 to 9 do begin
                            read(value);
                            ss_number[i] := value;
                            end;
                        writeln;
                        end;
                    end;
                if good_rec then begin
                    RPS_INSERT(SAS_ref);
                    check_for_error(SAS_ref^.status);
                    end;
                until not good_rec;

            {  Terminate scan.  }

            RPS_CLOSESCAN(SAS_ref);
            check_for_error(SAS_ref^.status);

            release(heap_ptr);

    end;  { add_employees }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section opens a scan on employee id.  It then sequentially     }
{    reads every employee and prints all employee information.           }
{                                                                        }
{------------------------------------------------------------------------}

procedure  list_employees;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        rec : data;
        i : integer;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for employees.  }

        new(SAS_ref);
        with SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := next;
            record_locking := false;
            key_id := 0;
            record_area_ref := RPS_PTR_LOAD(rec)
            end;
        RPS_OPENSCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  Fetch records and print informantion.  }

        RPS_FETCH(SAS_ref);
        check_for_error(SAS_ref^.status);
        while SAS_ref^.status <> eof_msg do begin
            with rec do begin
                write(employee_id:5,' ',first_name,' ',last_name,age:4,
                      sex:2,' ',birthday.month:2,'/',birthday.day:2,'/',
                      birthday.year:4,' ',citizenship,' ',salary:6:2,
                      manager_id:5,' ');
                for i := 1 to 9 do write(ss_number[i]);
                writeln;
                end;
            RPS_FETCH(SAS_ref);
            check_for_error(SAS_ref^.status);
            end;
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section opens a scan on employee id.  It then sequentially     }
{    reads every employee and prints all employee information.           }
{                                                                        }
{------------------------------------------------------------------------}

procedure  list_employees;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        rec : data;
        i : integer;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for employees.  }

        new(SAS_ref);
        with SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := next;
            record_locking := false;
            key_id := 0;
            record_area_ref := RPS_PTR_LOAD(rec)
            end;
        RPS_OPENSCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  Fetch records and print informantion.  }

        RPS_FETCH(SAS_ref);
        check_for_error(SAS_ref^.status);
        while SAS_ref^.status <> eof_msg do begin
            with rec do begin
                write(employee_id:5,' ',first_name,' ',last_name,age:4,
                      sex:2,' ',birthday.month:2,'/',birthday.day:2,'/',
                      birthday.year:4,' ',citizenship,' ',salary:6:2,
                      manager_id:5,' ');
                for i := 1 to 9 do write(ss_number[i]);
                writeln;
                end;
            RPS_FETCH(SAS_ref);
            check_for_error(SAS_ref^.status);
            end;
```

```
        { Terminate scans. }

        RPS_CLOSESCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        release(heap_ptr);

end; { list_employees }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section opens two scans.  One scan is opened on the salary index,  }
{    the other on the birthday index.  For each employee whose salary is     }
{    greater than or equal to $30,000, the employee's last name and birthday }
{    are printed.  All employees with the same birthday are found and their  }
{    full names are printed.  (Note this list of names will also include the }
{    names of the employees who make over $30,000.)                          }
{                                                                        }
{------------------------------------------------------------------------}

procedure  list_birthdays;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        salary_SAS_ref : ptr_SAS;
        salary_key : real;
        rec : data;
        birth_SAS_ref : ptr_SAS;
        birth_key  : date;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for salary.  }

        new(salary_SAS_ref);
        with salary_SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := direct;
            record_locking := false;
            key_id := 3;
            key_area_ref := RPS_PTR_LOAD(salary_key);
            record_area_ref := RPS_PTR_LOAD(rec);
            end;
        RPS_OPENSCAN(salary_SAS_ref);
        check_for_error(salary_SAS_ref^.status);

        {  Set up SAS and open scan for birthdays.  }

        new(birth_SAS_ref);
        with birth_SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := direct;
            record_locking := false;
```

```
        key_id := 2;
        key_area_ref := RPS_PTR_LOAD(birth_key);
        record_area_ref := RPS_PTR_LOAD(rec)
        end;
    RPS_OPENSCAN(birth_SAS_ref);
    check_for_error(birth_SAS_ref^.status);

    {  Fetch first salary greater than or equal to $30,000.  }

    salary_key := 30000.00;
    RPS_FETCH(salary_SAS_ref);
    check_for_error(salary_SAS_ref^.status);
    salary_SAS_ref^.access_mode := next;
    if no_error < salary_SAS_ref^.status then begin
        RPS_FETCH(salary_SAS_ref);
        check_for_error(salary_SAS_ref^.status);
        end;

    {  Fetch remaining salaries and birthdays.  }

    while salary_SAS_ref^.status <> eof_msg do begin
        with rec, birthday do begin
            writeln('Employee - ',last_name);
            writeln('  Birthday - ',month:2,'/',day:2,'/',year:4);
            birth_key := birthday;
            end;
        writeln('  Other Employees :');
        birth_SAS_ref^.access_mode := direct;
        RPS_FETCH(birth_SAS_ref);
        check_for_error(birth_SAS_ref^.status);
        birth_SAS_ref^.access_mode := next;
        while (rec.birthday = birth_key) and
              (birth_SAS_ref^.status <> eof_msg) do begin
                with rec do writeln('     ',first_name,last_name);
                RPS_FETCH(birth_SAS_ref);
                check_for_error(birth_SAS_ref^.status);
                end;
        RPS_FETCH(salary_SAS_ref);
        check_for_error(salary_SAS_ref^.status);
        end;

    {  Terminate scans.  }

    RPS_CLOSESCAN(birth_SAS_ref);
    check_for_error(birth_SAS_ref^.status);
    RPS_CLOSESCAN(salary_SAS_ref);
    check_for_error(salary_SAS_ref^.status);

    release(heap_ptr);

end;  { list_birthdays }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section opens two scans on the same index, the employee id.  It }
{    then sequentially reads every employee and prints the last name and }
{    age of the employee.  In addition, for each employee, the manager of }
{    that employee is found by a random fetch using employee id, and the }
{    manager's last name and age are printed.                            }
{                                                                        }
{------------------------------------------------------------------------}

procedure  list_ages;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        emp_SAS_ref : ptr_SAS;
        rec : data;
        mgr_SAS_ref : ptr_SAS;
        mgr_key : integer;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for employees.  }

        new(emp_SAS_ref);
        with emp_SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := next;
            record_locking := false;
            key_id := 0;
            record_area_ref := RPS_PTR_LOAD(rec)
            end;
        RPS_OPENSCAN(emp_SAS_ref);
        check_for_error(emp_SAS_ref^.status);

        {  Set up SAS and open scan for managers.  }

        new(mgr_SAS_ref);
        with mgr_SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := direct;
            record_locking := false;
            key_id := 0;
            key_area_ref := RPS_PTR_LOAD(mgr_key);
            record_area_ref := RPS_PTR_LOAD(rec)
```

```
            end;
        RPS_OPENSCAN(mgr_SAS_ref);
        check_for_error(mgr_SAS_ref^.status);

        {  Fetch records and print informantion.  }

        RPS_FETCH(emp_SAS_ref);
        check_for_error(emp_SAS_ref^.status);
        while emp_SAS_ref^.status <> eof_msg do begin
            with rec do writeln('Employee - ',last_name,'  Age - ',age:2);
            mgr_key := rec.manager_id;
            RPS_FETCH(mgr_SAS_ref);
            check_for_error(mgr_SAS_ref^.status);
            with rec do writeln('  Manager - ',last_name,'  Age - ',age:2);
            RPS_FETCH(emp_SAS_ref);
            check_for_error(emp_SAS_ref^.status);
            end;

        {  Terminate scans.  }

        RPS_CLOSESCAN(mgr_SAS_ref);
        check_for_error(mgr_SAS_ref^.status);
        RPS_CLOSESCAN(emp_SAS_ref);
        check_for_error(emp_SAS_ref^.status);

        release(heap_ptr);

end;  { list_ages }
```

```
{-------------------------------------------------------------------------}
{                                                                         }
{    This section lists all of the different salary amounts paid.  It also }
{    lists the names of all the poor unfortunate people who just made it  }
{    into the 50% tax bracket, namely those people earning $47,532.       }
{                                                                         }
{-------------------------------------------------------------------------}

procedure  list_salaries;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        key : real;
        rec : data;
        last_salary : real;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for salary.  }

        new(SAS_ref);
        with SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := read_only;
            access_mode := next;
            record_locking := false;
            key_id := 3;
            key_area_ref := RPS_PTR_LOAD(key);
            record_area_ref := RPS_PTR_LOAD(rec);
            end;
        RPS_OPENSCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  List salaries and unfortunate souls.  }

        last_salary := 0.00;
        RPS_POSITION(SAS_ref);
        check_for_error(SAS_ref^.status);
        while SAS_ref^.status <> eof_msg do begin
            if key <> last_salary then begin
                writeln(' Salary Paid - ',key:6:3);
                last_salary := key;
                end;
            if key = 47532.00 then begin
                SAS_ref^.access_mode := current;
                RPS_FETCH(SAS_ref);
                check_for_error(SAS_ref^.status);
```

```
                writeln(' Poor Unfortunate - ',rec.first_name,rec.last_name);
                SAS_ref^.access_mode := next;
                end;
            RPS_POSITION(SAS_ref);
            check_for_error(SAS_ref^.status);
            end;

        { Terminate scans. }

        RPS_CLOSESCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        release(heap_ptr);

end;  { list_salaries }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section removes all employees whose salary is greater than or  }
{    equal to $50,000.                                                   }
{                                                                        }
{------------------------------------------------------------------------}

procedure  remove_highly_paid;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        key : real;
        rec : data;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for salary.  }

        new(SAS_ref);
        with SAS_ref^ do begin
           scan_id := -1;
           filename_ref := RPS_PTR_LOAD(name_of_file);
           file_use := shared;
           process_required := modify;
           access_mode := direct;
           record_locking := false;
           key_id := 3;
           key_area_ref := RPS_PTR_LOAD(key);
           record_area_ref := RPS_PTR_LOAD(rec);
           end;
        RPS_OPENSCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  Fetch first salary greater than or equal to $50,000.  }

        key := 50000.00;
        RPS_FETCH(SAS_ref);
        check_for_error(SAS_ref^.status);
        SAS_ref^.access_mode := next;
        if no_error < SAS_ref^.status then RPS_FETCH(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  Delete all records with salary greater than or equal to $50,000.  }

        while SAS_ref^.status <> eof_msg do begin
            RPS_DELETE(SAS_ref);
            check_for_error(SAS_ref^.status);
            RPS_FETCH(SAS_ref);
            check_for_error(SAS_ref^.status);
```

```
            end;

        { Terminate scans. }

        RPS_CLOSESCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        release(heap_ptr);

end;  { delete_highly_paid }
```

```
{-------------------------------------------------------------------------}
{                                                                         }
{    This section increases all remaining employee salaries by $4,000.    }
{                                                                         }
{-------------------------------------------------------------------------}

procedure  give_raises;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        SAS_ref : ptr_SAS;
        rec : data;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up SAS and open scan for employees.  }

        new(SAS_ref);
        with SAS_ref^ do begin
            scan_id := -1;
            filename_ref := RPS_PTR_LOAD(name_of_file);
            file_use := shared;
            process_required := modify;
            access_mode := next;
            record_locking := false;
            key_id := 0;
            record_area_ref := RPS_PTR_LOAD(rec)
            end;
        RPS_OPENSCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        {  Fetch records and update salaries.  }

        RPS_FETCH(SAS_ref);
        check_for_error(SAS_ref^.status);
        while SAS_ref^.status <> eof_msg do begin
            rec.salary := rec.salary + 4000.00;
            RPS_UPDATE(SAS_ref);
            check_for_error(SAS_ref^.status);
            RPS_FETCH(SAS_ref);
            check_for_error(SAS_ref^.status);
            end;

        {  Terminate scan.  }

        RPS_CLOSESCAN(SAS_ref);
        check_for_error(SAS_ref^.status);

        release(heap_ptr);
```

```
end;  { give_raises }
```

```
{-------------------------------------------------------------------------}
{                                                                         }
{    This section erases all employee records.  After the file has been   }
{    emptied, new employee records are added to the file.                 }
{                                                                         }
{-------------------------------------------------------------------------}

procedure  form_new_employee_list;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        FAS_ref : ptr_FAS;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up FAS.  }

        new(FAS_ref);
        FAS_ref^.filename_ref := RPS_PTR_LOAD(name_of_file);

        {  Erase file.  }

        RPS_ERASE(FAS_ref);
        check_for_error(FAS_ref^.status);

        {  Add new employees.  }

        add_employees;

        release(heap_ptr);

 end;  { form_new_employee_list }
```

```
{------------------------------------------------------------------------------}
{                                                                              }
{    This section destroys the employee file.                                  }
{                                                                              }
{------------------------------------------------------------------------------}

procedure  destroy_employee_file;

var
        heap_ptr : ptr_rps_data;
        name_of_file : string[30];
        FAS_ref : ptr_FAS;
        rec : data;

begin
        mark(heap_ptr);

        name_of_file := '/PROFILE/PAYROLL/EMPLOYEES';

        {  Set up FAS.  }

        new(FAS_ref);
        FAS_ref^.filename_ref := RPS_PTR_LOAD(name_of_file);

        {  Destroy file.  }

        RPS_DESTROY(FAS_ref);
        check_for_error(FAS_ref^.status);

        release(heap_ptr);

end;  { destroy_employee_file }
```

```
{------------------------------------------------------------------------}
{                                                                        }
{    This section is the main body of the program.                       }
{                                                                        }
{------------------------------------------------------------------------}

begin
        set_up_storage;
        make_employee_file;
        list_file_characteristics;
        add_employees;
        list_employees;
        list_birthdays;
        list_ages;
        list_salaries;
        remove_highly_paid;
        give_raises;
        list_employees;
        form_new_employee_list;
        list_employees;
        destroy_employee_file;
end.
```

@@@@ Appendix F

@@@ RPS Interface

```
unit RPS;

intrinsic code 25 data 26;

Interface

type
        rps_byte = 0..255;
        rps_data = packed array [0..0] of rps_byte;
        ptr_rps_data = ^ rps_data;
        rps_count = array [0..1] of integer;

        modes_of_processing = (read_only, modify);
        modes_of_access = (first, last, direct, current, next, prior);
        type_of_records = (fixed_length, variable_length);
        type_of_segs = (seg_ints16, seg_intu16,
                        seg_ints32, seg_intu32,
                        seg_ints64,  seg_intu64,
                        seg_real32, seg_real64,
                        seg_string, seg_packchar,
                        seg_packbyte, seg_unknown);
        type_of_uses = (shared, exclusive);
        sort_order = (ascending, descending);
        release_priorities = array [0..10] of integer;
```

```
ptr_IS = ^IS;
IS =
    record                                              {  default values  }
        key_size : integer;                             {        20        }
        record_size : integer                           {       100        }
        num_files : integer;                            {         1        }
        num_scans : integer;                            {         2        }
        num_indexes : integer;                          {         4        }
        num_segs : integer;                             {        16        }
        num_512_buffs : integer;                        {         8        }
        num_1024_buffs : integer;                       {         0        }
        num_1536_buffs : integer;                       {         0        }
        num_2048_buffs : integer;                       {         0        }
        rel_priority : release_priorities;              {  0,10,10,...,10  }
        write_threshold : integer;                      {         1        }
        extend_size : integer;                          {         4        }
        status : integer;
    end;

ptr_SD = ^SD;
SD =
    record
        pos_in_record : integer;
        seg_size : integer;
        seg_type : type_of_segs;
        sort_direction : sort_order;
        next_SD_ref : ptr_SD;
    end;

ptr_KD = ^KD;
KD =
    record
        dupls_allowed : boolean;
        nulls_allowed : boolean;
        nulls_kept : boolean;
        index_page_size : integer;
        next_KD_ref : ptr_KD;
        SD_ref : ptr_SD;
    end;

ptr_FAS = ^FAS;
FAS =
    record
        filename_ref : ptr_rps_data;
        record_size : integer;
        record_type : type_of_records;
        data_page_size : integer;
        KD_ref : ptr_KD;
        status : integer;
    end;
```

```
ptr_SAS = ^SAS;
SAS =
    record
        scan_id : integer;
        filename_ref : ptr_rps_data;
        file_use : type_of_uses;
        process_required : modes_of_processing;
        access_mode : modes_of_access;
        record_locking : boolean;
        key_id : integer;
        key_area_ref : ptr_rps_data;
        record_area_ref : ptr_rps_data;
        status : integer;
    end;

ptr_SIS = ^SIS;
SIS =
    record
        filename_ref : ptr_rps_data;
        key_id : integer;
        seg_id : integer;
        pos_in_key : integer;
        pos_in_record : integer;
        seg_size : integer;
        seg_type : type_of_segs;
        sort_direction : sort_order;
        status : integer;
    end;

ptr_KIS = ^KIS;
KIS =
    record
        filename_ref : ptr_rps_data;
        key_id : integer;
        dupls_allowed : boolean;
        nulls_allowed : boolean;
        nulls_kept : boolean;
        index_page_size : integer;
        num_segs : integer;
        key_count : rps_count;
        index_depth : integer;
        status : integer;
    end;
```

```
        ptr_FIS = ^FIS;
        FIS =
            record
                filename_ref : ptr_rps_data;
                record_size : integer;
                record_type : type_of_records;
                record_count : rps_count;
                data_page_size : integer;
                num_indexes : integer;
                status : integer;
            end;

function    RPS_PTR_LOAD (var data_ref) : ptr_rps_data;

procedure   RPS_INITIALIZE (IS_ref : ptr_IS);

procedure   RPS_CREATE (FAS_ref : ptr_FAS);

procedure   RPS_ERASE (FAS_ref : ptr_FAS);

procedure   RPS_DESTROY (FAS_ref : ptr_FAS);

procedure   RPS_OPENSCAN (SAS_ref : ptr_SAS);

procedure   RPS_CLOSESCAN (SAS_ref : ptr_SAS);

procedure   RPS_FETCH (SAS_ref : ptr_SAS);

procedure   RPS_POSITION (SAS_ref : ptr_SAS);

procedure   RPS_INSERT (SAS_ref : ptr_SAS);

procedure   RPS_UPDATE (SAS_ref : ptr_SAS);

procedure   RPS_DELETE (SAS_ref : ptr_SAS);

procedure   RPS_SEGINFO (SIS_ref : ptr_SIS);

procedure   RPS_KEYINFO (KIS_ref : ptr_KIS);

procedure   RPS_FILEINFO (FIS_ref : ptr_FIS);
```

```
unit RPS_REPAIR;

intrinsic code 27;

Interface

uses  rps;

type
        ptr_DRS = ^DRS;
        DRS =
            record
                damaged_filename_ref : ptr_rps_data;
                data_file_missing : boolean;
                data_file_not_rps : boolean;
                index_file_missing : boolean;
                index_file_not_rps : boolean;
                FAS_damaged : boolean;
                KD_damaged : packed array [0..7] of boolean;
                SD_damaged : packed array [0..7,0..7] of boolean;
                data_corrupted : boolean;
                index_corrupted : packed array [0..7] of boolean;
                status : integer;
            end;

        ptr_RBS = ^RBS;
        RBS =
            record
                damaged_filename_ref : ptr_rps_data;
                new_filename_ref : ptr_rps_data;
                discard_filename_ref : ptr_rps_data;
                corrected_FAS_ref : ptr_FAS;
                discard_limit : rps_count;
                status : integer;
            end;

procedure  RPS_DAMAGE_REPORT (DRS_ref : ptr_DRS);

procedure  RPS_DMG_SEGINFO (SIS_ref : ptr_SIS);

procedure  RPS_DMG_KEYINFO (KIS_ref : ptr_KIS);

procedure  RPS_DMG_FILEINFO (FIS_ref : ptr_FIS);

procedure  RPS_REBUILD (RBS_ref : ptr_RBS);
```

@@@@ RPS Reference Card

@@@ RPS Interface

```
unit RPS; intrinsic code 25 data 26;

Types
        rps_byte = 0..255;
        rps_data = packed array [0..0] of rps_byte;
        ptr_rps_data = ^ rps_data;
        rps_count = array [0..1] of integer;

        modes_of_processing = (read_only, modify);
        modes_of_access = (first, last, direct, current, next, prior);
        type_of_records = (fixed_length, variable_length);
        type_of_segs = (seg_ints16, seg_intu16,
                        seg_ints32, seg_intu32,
                        seg_ints64,  seg_intu64,
                        seg_real32, seg_real64,
                        seg_string, seg_packchar,
                        seg_packbyte, seg_unknown);
        type_of_uses = (shared, exclusive);
        sort_order = (ascending, descending);
        release_priorities = array [0..10] of integer;

        ptr_IS = ^IS;
        IS =
            record                                          {  default values  }
                key_size : integer;                         {        20        }
                record_size : integer                       {       100        }
                num_files : integer;                        {         1        }
                num_scans : integer;                        {         2        }
                num_indexes : integer;                      {         4        }
                num_segs : integer;                         {        16        }
                num_512_buffs : integer;                    {         8        }
                num_1024_buffs : integer;                   {         0        }
                num_1536_buffs : integer;                   {         0        }
                num_2048_buffs : integer;                   {         0        }
                rel_priority : release_priorities;          {  0,10,10,...,10  }
                write_threshold : integer;                  {         1        }
                extend_size : integer;                      {         4        }
                status : integer;
            end;

        ptr_SD = ^SD;
        SD =
```

```
        record
            pos_in_record : integer;
            seg_size : integer;
            seg_type : type_of_segs;
            sort_direction : sort_order;
            next_SD_ref : ptr_SD;
        end;

    ptr_KD = ^KD;
    KD =
        record
            dupls_allowed : boolean;
            nulls_allowed : boolean;
            nulls_kept : boolean;
            index_page_size : integer;
            next_KD_ref : ptr_KD;
            SD_ref : ptr_SD;
        end;

    ptr_FAS = ^FAS;
    FAS =
        record
            filename_ref : ptr_rps_data;
            record_size : integer;
            record_type : type_of_records;
            data_page_size : integer;
            KD_ref : ptr_KD;
            status : integer;
        end;

    ptr_SAS = ^SAS;
    SAS =
        record
            scan_id : integer;
            filename_ref : ptr_rps_data;
            file_use : type_of_uses;
            process_required : modes_of_processing;
            access_mode : modes_of_access;
            record_locking : boolean;
            key_id : integer;
            key_area_ref : ptr_rps_data;
            record_area_ref : ptr_rps_data;
            status : integer;
        end;
```

```
ptr_SIS = ^SIS;
SIS =
    record
        filename_ref : ptr_rps_data;
        key_id : integer;
        seg_id : integer;
        pos_in_key : integer;
        pos_in_record : integer;
        seg_size : integer;
        seg_type : type_of_segs;
        sort_direction : sort_order;
        status : integer;
    end;

ptr_KIS = ^KIS;
KIS =
    record
        filename_ref : ptr_rps_data;
        key_id : integer;
        dupls_allowed : boolean;
        nulls_allowed : boolean;
        nulls_kept : boolean;
        index_page_size : integer;
        num_segs : integer;
        key_count : rps_count;
        index_depth : integer;
        status : integer;
    end;

ptr_FIS = ^FIS;
FIS =
    record
        filename_ref : ptr_rps_data;
        record_size : integer;
        record_type : type_of_records;
        record_count : rps_count;
        data_page_size : integer;
        num_indexes : integer;
        status : integer;
    end;
```

RPS Calls

```
idxpgszltlim =  -1306;
idxpgszgtlim =  -1307;
noptrtokd    =  -1308;
noptrtosd    =  -1309;
nulluseinvld =  -1310;
segposinvld  =  -1311;
segsizinvld  =  -1312;
numkeygtlim  =  -1313;
numseggtlim  =  -1314;
keypagszconf =  -1315;
filnotrps    =  -1401;
filnamgtlim  =  -1402;
nodatafile   =  -1403;
noindexfile  =  -1404;
corruptdata  =  -1405;
corruptindex =  -1406;
keyidinvld   =  -1407;
segidinvld   =  -1408;
scanidinvld  =  -1409;
numfilegtlim =  -1410;
numindxgtlim =  -1411;
numscangtlim =  -1412;
crosinv      =  -1501;
recinuse     =  -1502;
reclocked    =  -1503;
procnotalwd  =  -1504;
nopathtorec  =  -1505;
noduplalwd   =  -1550;    { -1550,...,-1557 low digit indicates index }
nonullalwd   =  -1560;    { -1560,...,-1567 low digit indicates index }
indxtoolarge =  -1570;    { -1570,...,-1577 low digit indicates index }
leveltoodeep =  -1580;    { -1580,...,-1587 low digit indicates index }
```

```
ptr_RBS = ^RBS;
RBS =
    record
        damaged_filename_ref : ptr_rps_data;
        new_filename_ref : ptr_rps_data;
        discard_filename_ref : ptr_rps_data;
        corrected_FAS_ref : ptr_FAS;
        discard_limit : rps_count;
        status : integer;
    end;
```

RPS_REPAIR Calls

```
procedure  RPS_DAMAGE_REPORT (DRS_ref : ptr_DRS);

procedure  RPS_DMG_SEGINFO (SIS_ref : ptr_SIS);

procedure  RPS_DMG_KEYINFO (KIS_ref : ptr_KIS);

procedure  RPS_DMG_FILEINFO (FIS_ref : ptr_FIS);

procedure  RPS_REBUILD (RBS_ref : ptr_RBS);
```

@@@ RPS Status Returns

```
soffile      =   1003;
eoffile      =   1002;
recnotfound  =   1001;
success      =      0;
nomemavail   =  -1101;
nobuffavail  =  -1102;
volumefull   =  -1103;
alreadyinit  =  -1201;
invldkeysiz  =  -1202;
invldrecsiz  =  -1203;
invldnfile   =  -1204;
invldnscan   =  -1205;
invldnindex  =  -1206;
invldnseg    =  -1207;
invldnbuff   =  -1208;
invldpriorty =  -1209;
invldwthresh =  -1210;
invldextsiz  =  -1211;
recsizltlim  =  -1301;
recsizgtlim  =  -1302;
recpagszconf =  -1303;
datpgszltlim =  -1304;
datpgszgtlim =  -1305;
```

```
function   RPS_PTR_LOAD (var data_ref) : ptr_rps_data;

procedure  RPS_INITIALIZE (IS_ref : ptr_IS);

procedure  RPS_CREATE (FAS_ref : ptr_FAS);

procedure  RPS_ERASE (FAS_ref : ptr_FAS);

procedure  RPS_DESTROY (FAS_ref : ptr_FAS);

procedure  RPS_OPENSCAN (SAS_ref : ptr_SAS);

procedure  RPS_CLOSESCAN (SAS_ref : ptr_SAS);

procedure  RPS_FETCH (SAS_ref : ptr_SAS);

procedure  RPS_POSITION (SAS_ref : ptr_SAS);

procedure  RPS_INSERT (SAS_ref : ptr_SAS);

procedure  RPS_UPDATE (SAS_ref : ptr_SAS);

procedure  RPS_DELETE (SAS_ref : ptr_SAS);

procedure  RPS_SEGINFO (SIS_ref : ptr_SIS);

procedure  RPS_KEYINFO (KIS_ref : ptr_KIS);

procedure  RPS_FILEINFO (FIS_ref : ptr_FIS);
```

@@@ RPS_REPAIR Interface

```
unit RPS_REPAIR; intrinsic code 27;

Types
        ptr_DRS = ^DRS;
        DRS =
            record
                damaged_filename_ref : ptr_rps_data;
                data_file_missing : boolean;
                data_file_not_rps : boolean;
                index_file_missing : boolean;
                index_file_not_rps : boolean;
                FAS_damaged : boolean;
                KD_damaged : packed array [0..7] of boolean;
                SD_damaged : packed array [0..7,0..7] of boolean;
                data_corrupted : boolean;
                index_corrupted : packed array [0..7] of boolean;
                status : integer;
            end;
```